Extendiendo Darmok, un Sistema de Planificación Basada en Casos, mediante Árboles de Comportamiento

Proyecto Fin de Máster en Sistemas Inteligentes

Máster en Investigación en Informática
Facultad de Informática
Universidad Complutense de Madrid

Autor: Ricardo Juan Palma Durán
Director: Pedro Antonio González Calero

Curso 2009/2010

Extendiendo Darmok, un Sistema de Planificación Basada en Casos, mediante Árboles de Comportamiento

Ricardo Juan Palma Durán
Universidad Complutense de Madrid
odracirnumira@gmail.com

Agradecimientos

Desde un principio querría dar las gracias a todas aquellas personas que han hecho posible la realización de este proyecto. A Santiago Ontañón Villar, por ofrecerme su ayuda de forma continua hasta el último momento, y por llevar a sus espaldas la dura tarea de definir el dominio MakeMePlayMe de StarCraft. A Pedro y Marco Gómez Martín, gracias a cuyas ideas fue posible la implementación del framework de desarrollo de árboles de comportamiento. Y a Pedro Antonio González Calero, director de este proyecto, el cual, a pesar de las incontables tareas que como investigador profesional le agobian, pudo sacar tiempo y ganas de guiar a este pobre hombre descarriado.

Abstract

The combination of learning and planning has proved an effective solution for real-time strategy games. Nevertheless, learning hierarchical plans from expert traces also has its limitations, regarding the number of training traces required, and the absence of mechanisms for rapidly reacting to high priority goals. We propose to bring the game designer back into the loop, by allowing him to explicitly inject decision making knowledge, in the form of behavior trees, to complement the knowledge obtained from the traces. By providing a natural mechanism for designers to inject knowledge into the plan library, we intend to integrate the best of both worlds: learning from traces and hard-coded rules.

In this work we extend a well known case-based planning system, Darmok, to overcome some of its most important flaws. Darmok, initially designed to play real-time strategy games, does have some problems concerning its reactivity capabilities. In particular, it can be either too reactive when executing some plans, or too non-reactive when executing some low level actions, which makes its performance greately deteriorate in some scenarios. By using expert knowledge in the form of behaviour trees, we intend to overcome these problems.

Key words: Strategic Decision Making, Tactical Decision Making, Planning, Behavior Modeling, Behaviour Tree, Darmok, Case-Based Reasoning.

Resumen

La combinación de aprendizaje automático y planificación ha demostrado ser una buena solución para los juegos de estrategia en tiempo real. Sin embargo, aprender planes jerárquicos a partir de trazas (demostraciones) de expertos tiene también sus limitaciones, en particular el número de trazas de entrenamiento necesitadas, así como la ausencia de mecanismos para reaccionar rápidamente ante objetivos de alta prioridad. En este trabajo proponemos que el diseñador defina de manera explícita conocimiento para la toma de decisiones en forma de árboles de comportamiento, para así complementar el conocimiento obtenido a partir de las trazas. Proporcionando un mecanismo natural para que los diseñadores inyecten conocimiento en la librería de planes, pretendemos integrar lo mejor de ambos enfoques: aprendizaje de trazas y reglas codificadas a mano.

En este trabajo extendemos un sistema de planificación basada en casos, Darmok, para resolver algunos de sus problemas más importantes. Darmok, que fue diseñado para jugar a juegos de estrategia en tiempo real, tiene algunas deficiencias en lo que respecta a su capacidad de reactividad. En concreto, puede ser o demasiado reactivo cuando ejecuta algunos planes, o demasiado no-reactivo en la ejecución de algunas acciones de bajo nivel, lo cual en su conjunto hace que su rendimiento se deteriore en algunos escenarios bastante frecuentes. Mediante el uso de conocimiento experto en forma de árboles de comportamiento, pretendemos solventar estos problemas.

Palabras clave: Toma de Decisiones a Nivel Estratégico, Toma de Decisiones a Nivel Táctico, Planificación, Modelado de Comportamiento, Árbol de Comportamiento, Darmok, Razonamiento Basado en Casos.

Índice general

Agradecimientos iii

Abstract v

Resumen vii

Introducción xv

1. Planificación Basada en Casos, Darmok 1
- 1.1. Razonamiento Basado en Casos . . . 1
- 1.2. Planificación Basada en Casos . . . 2
- 1.3. Darmok, un Sistema CBP para Juegos de Estrategia . . . 3
  - 1.3.1. Representación de Planes . . . 4
  - 1.3.2. Adquisición de Planes . . . 5
  - 1.3.3. Recuperación de Planes . . . 6
  - 1.3.4. Fase de Expansión y Ejecución En Linea . . . 6
  - 1.3.5. Adaptación . . . 7

2. Árboles de Comportamiento 9
- 2.1. Mecánica de Funcionamiento . . . 10
- 2.2. Contexto de Ejecución . . . 11
- 2.3. Tipos de Tareas . . . 13
  - 2.3.1. Tareas de Bajo Nivel . . . 13
  - 2.3.2. Tareas Compuestas . . . 13
  - 2.3.3. Decoradores . . . 17
- 2.4. Reutilización de Árboles . . . 18

3. Combinando Darmok con BTs 21
- 3.1. Capa Táctica de Bajo Nivel . . . 23
- 3.2. Capa Táctica para la Gestión de Planes Objetivo . . . 24
- 3.3. Arquitectura Global . . . 25
- 3.4. Recuperación de Árboles . . . 27
- 3.5. Escenario . . . 28
  - 3.5.1. Escenario de Acciones de Bajo Nivel . . . 29
  - 3.5.2. Escenario de Plan Objetivo . . . 29

4. Java Behaviour Trees 31
- 4.1. Características Principales de JBT . . . 31
  - 4.1.1. Modelo Conducido por Ticks . . . 31
  - 4.1.2. Modelo Independiente de Ejecución . . . 32
  - 4.1.3. Arquitectura Global . . . 33

4.1.4. Modelo de Árboles de Comportamiento 33
4.2. JBT Editor 34

5. Estudio Experimental **35**
5.1. Dominio, StarCraft 35
5.2. Comunicación con StarCraft 36
5.2.1. Comunicación en Java 37
5.3. Diseño de los Experimentos 38
5.3.1. Definición del Dominio 38
5.3.2. Experimentos 39
5.4. Experimento 1 40
5.5. Experimento 2 44
5.6. Experimento 3 47

6. Conclusiones y Trabajo Futuro **51**

A. JBT, Guía del Usuario **53**
A.1. Introduction 53
A.2. JBT, an Overview 54
A.2.1. Model Driven by Ticks 54
A.2.2. Model Independent from Execution 55
A.2.3. Architecture 55
A.2.4. BT Model 56
A.2.4.1. Execution Context 56
A.2.4.2. Native Tasks 59
A.3. Defining Actions And Conditions 60
A.4. Implementing Actions And Conditions 65
A.5. Creating BTs with JBT Editor 73
A.6. Creating a Java Declaration of the BTs 78
A.7. Running the Bthaviour Trees 81

Índice de figuras

1.1. El ciclo del razonamiento basado en casos 2
1.2. El ciclo OLCBP 4
1.3. Extracción de casos en Darmok a partir de una traza 5
1.4. Fase de ejecución en Darmok 7

2.1. Estructura de un nodo de un árbol de comportamiento 11
2.2. El contexto de ejecución 12
2.3. Pseudo-implementación del nodo *secuencia* 15
2.4. Un árbol de comportamiento simple con un nodo *secuencia* . . . 16
2.5. Un árbol de comportamiento simple con un nodo *selector* 16
2.6. Un árbol de comportamiento para el comportamiento *Entrar en habitación* 17
2.7. El árbol de comportamiento de la figura 2.6, simplificado 19

3.1. Capa táctica de bajo nivel 25
3.2. Capa táctica de alto nivel 26
3.3. Árbol de comportamiento para la acción *ConvocarTormentaPsionica* 29

4.1. Arquitectura global de JBT 33
4.2. JBT Editor 34

5.1. Una partida de StarCraft, donde se enfrentan dos ejércitos . . . 36
5.2. Integración de BWAPI con StarCraft 37
5.3. Proxy Bot para la comunicación Java con Starcraft 38
5.4. Árbol de comportamiento para la acción *Atacar*. Con un borde negro remarcado se señalan las acciones y condiciones dependientes del dominio. 41
5.5. Árbol de comportamiento para la acción *Moverse*. Con un borde negro remarcado se señalan las acciones y condiciones dependientes del dominio 42
5.6. Escenario del experimento 1 43
5.7. Árbol de comportamiento para la acción *Atacar*. Ahora el árbol contempla la presencia de búnkeres cercanos 45
5.8. Escenario del experimento 2 46
5.9. Árbol de comportamiento para la acción *Atacar* del buitre Terran 48
5.10. Escenario del experimento 3 49

A.1. Overview of the BT architecture 55
A.2. a simple behaviour tree 56
A.3. a complex behaviour tree 57
A.4. JBT Editor after being opened 74

A.5. The Nodes Navigator after loading the domain file 75
A.6. The dialog for editing the input parameters of the AttackMove action 75
A.7. A parameter for which an actual value is provided 75
A.8. A parameter whose value will be retrieved from the context . . . 75
A.9. Initial tree for the Terran Marine behaviour 76
A.10. Selecting a guard for the *Attack* node 77
A.11. The input parameter of *Attack* 77
A.12. The BT for the Standard Patrol behaviour 78

Índice de tablas

5.1. Resultados del experimento 1 43
5.2. Resultados del experimento 2 46
5.3. Resultados del experimento 3 49

Introducción

El desarrollo de videojuegos ha experimentado una gran evolución durante las últimas dos décadas, en especial gracias a la aparición de equipos informáticos con un hardware más potente que ha permitido dar soporte a gráficos y sonido mucho más realistas. Al mismo tiempo, la inteligencia artificial (IA) en videojuegos está adquiriendo cada vez un papel más relevante. Una IA de alta calidad supone incrementar el valor de entretenimiento de un juego, y como consecuencia, se alcanza un mayor número de ventas. Debe tenerse en cuenta que el ámbito de los videojuegos no se reduce solamente al lúdico. Cada vez más, los videojuegos se están usando como herramientas de educación y entrenamiento en todo tipo de áreas (militar, económica o universitaria, por nombrar unos ejemplos).

El problema del desarrollo de IAs en videojuegos es que, por lo general, éstos simulan entornos virtuales donde el espacio de soluciones es inmanejable, y en los que además se tiene que tomar decisiones en tiempo real. En este contexto, las técnicas clásicas de búsqueda mediante la exploración del espacio de soluciones son inaplicables, hecho que ha motivado la aparición de técnicas alternativas capaces de hacer frente a este doble requerimiento. Los juegos de estrategia en tiempo real son un claro ejemplo de ello: tienen espacios de decisión enormes, son dominios de competición entre adversarios, son no-determinísticos, no son completamente observables y además suele ser difícil definir postcondiciones para las acciones del dominio (las acciones no siempre tienen éxito, y pueden darse interacciones inimaginablemente complejas que son difícilmente modelables) [25]. En Wargus, un juego de estrategia en tiempo real parecido al conocido Warcraft 2, por ejemplo, poco después del inicio de una partida el espacio de decisión alcanza un tamaño del orden de las miles de acciones [26], lo cual da una idea de la complejidad de este tipo de dominios.

Tradicionalmente, esta situación ha dado lugar a que los desarrolladores acaben codificando manualmente la IA de los videojuegos. Sin embargo, el esfuerzo requerido para la codificación manual de una IA capaz de interaccionar con jugadores humanos suele ser muy grande. Durante el proceso, además, los desarrolladores suelen cometer errores que degradan la calidad de la IA que finalmente verá la luz. Más aún, las IAs codificadas manualmente carecen del componente adaptativo que se requiere cuando se juega contra seres humanos. En este sentido, es fácil que los jugadores exploten tanto la estaticidad como las deficiencias de las IAs, pudiendo derrotarlas fácilmente. Dentro de los sistemas de codificación manual, los *árboles de comportamiento* han adquirido cierta popularidad en los últimos años, debido principalmente a su gran simplicidad y capacidad de reutilización. Mediante el uso de árboles de comportamiento se reduce significativamente la dificultad a la hora de diseñar inteligencias complejas.

El desarrollo de técnicas de IA más avanzadas aplicables al campo de los

videojuegos tiene como objetivos principales ayudar a los desarrolladores a la creación de IAs de una manera eficiente, y a que éstas tengan un carácter adaptativo que incremente su realismo y capacidad. El empleo de técnicas de *aprendizaje automático* facilita a los desarrolladores la construcción de inteligencias siguiendo una filosofía de *demostración mediante ejemplos*, permitiendo diseñar de una manera mucho más sencilla IAs igualmente complejas y realistas. La adopción de técnicas de aprendizaje automático embebidas en el propio videojuego permite que las IAs evolucionen con el paso del tiempo, adaptándose a entornos dinámicos y situaciones desconocidas. En este contexto se han aplicado, con un éxito parcial, técnicas como redes neuronales [10, 33], planificación basada en casos [23], o más recientemente el *scripting dinámico* [32].

En este estudio proponemos extender un sistema completo de planificación basada en casos orientado a juegos de estrategia en tiempo real, Darmok [23], mediante la inclusión de conocimiento experto en la forma de árboles de comportamiento, con el propósito de eliminar algunas de las carencias más importantes de dicho sistema. Darmok consta de problemas de reactividad que le impiden reaccionar adecuadamente en ciertos escenarios frecuentes en juegos de estrategia. Los árboles de comportamiento, por otro lado, son estructuras innatamente reactivas, que permiten un control preciso de los agentes que manejan. Mediante la combinación de ambos pretendemos resolver esta carencia de Darmok.

La estructura de este trabajo es la que sigue. En el capítulo 1 presentamos el sistema Darmok como ejemplo de arquitectura de planificación basada en casos. El capítulo 2 describe en detalle la técnica de los árboles de comportamiento. A continuación, en el capítulo 3 se explican algunas de las deficiencias de Darmok y cómo el uso combinado de árboles de comportamiento y planificación basada en casos pueden resolverlos. Posteriormente, en el capítulo 4 se describe brevemente JBT, el framework de desarrollo de árboles de comportamiento que implementamos para este proyecto. Por último, los capítulos 5 y 6 recogen el estudio experimental llevado a cabo así como las conclusiones y propuestas de trabajo futuro.

Capítulo 1

Planificación Basada en Casos, Darmok

1.1. Razonamiento Basado en Casos

El *razonamiento basado en casos* (CBR[1]) [1, 11] es una metodología de resolución de problemas basada en la reutilización de conocimiento previo, almacenado en forma de *casos*, para hacer frente a problemas similares a los encontrados en el pasado. Grosso modo, un caso representa una experiencia usada en el pasado para la resolución de un determinado problema. Cuando un sistema CBR encuentra un nuevo problema, analiza su *base de casos* en búsqueda de alguno que fuera usado en circunstancias similares a la actual (fase de recuperación), con la idea de reutilizarlo. Esta reutilización, sin embargo, requiere adaptar el caso encontrado al problema actual, ya que el caso recuperado, en su momento, no fue usado para resolver exactamente el mismo problema (fase de reutilización). Posteriormente, se evalúa la calidad de la solución propuesta, y se corrige en caso de ser necesario (fase de revisión). Finalmente, la solución obtenida se almacena en la base de casos como un nuevo caso, para así aumentar el conocimiento disponible (fase de retención). La figura 1.1 muestra el ciclo clásico de un sistema CBR, donde se aprecian las cuatro fases mencionadas.

Si bien el ciclo CBR actualmente está sólidamente establecido, dependiendo del dominio de aplicación sus distintas partes pueden diseñarse de maneras muy distintas entre sí. En particular, representaciones del dominio muy diferentes pueden dar lugar a fases de recuperación o adaptación muy diferentes. Otras características, como la organización de la información en la base de casos o las métricas de similitud usadas en la fase de recuperación pueden conllevar amplias diferencias en lo que a eficiencia de cómputo se refiere.

El CBR ha sido aplicado exitosamente en la resolución de todo tipo de problemas, en particular en dominios poco formalizados en los que el aprendizaje automático juega un papel importante, ya que se pueden proponer soluciones a problemas incluso cuando no se comprenden con todo detalle.

Algunos de los campos en los que se ha empleado CBR son: diagnóstico (sistemas Protos [4] para enfermedades del oído, y Caseline [16] para la detección y solución de fallos en Boeing 747); enseñanza (sistemas Hypo [30] para la enseñanza de razonamiento legal, y Spiel [6] para el aprendizaje de habilidades sociales); razonamiento con adversarios (Judge [3] un sistema que emula a un

[1]De sus siglas en inglés, *case-based reasoning*.

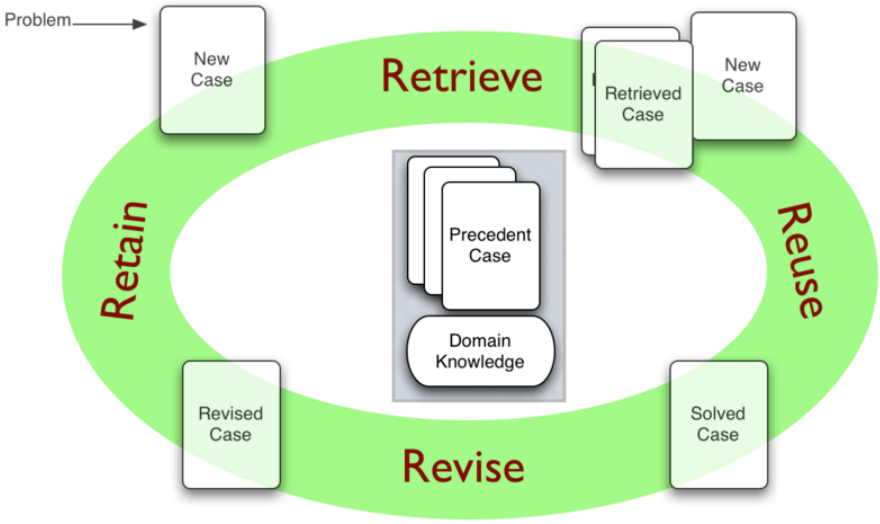

Figura 1.1: El ciclo del razonamiento basado en casos

juez que sentencia a criminales); soporte técnico (Smart [22], que ayuda a los técnicos de soporte en la resolución de incidencias de sus usuarios); o resolución de problemas y planificación (como Bolero [15], que ayuda a la construcción de pruebas médicas para la obtención de un diagnóstico), entre otros.

1.2. Planificación Basada en Casos

Una de las áreas donde el CBR ha sido aplicado con resultados positivos es el de la resolución de problemas mediante técnicas de planificación automática. La planificación automática de tareas [7] tiene como objetivo la resolución de problemas (objetivos) mediante la generación de secuencias específicas de instrucciones. Tradicionalmente, las técnicas de planificación automática se han basado en explorar de forma inteligente el espacio de todas las posibles secuencias de instrucciones, encontrando una que logre satisfacer el objetivo inicial. Debido a la inmensidad del espacio de posibles secuencias de acciones en dominios reales, más recientemente se han propuesto otras alternativas, como la planificación automática mediante redes de tareas jerárquicas (HTNs) [21]. La planificación HTN se basa en descomponer el objetivo original en una jerarquía de objetivos, cada uno de los cuales se resuelve de forma independiente, acelerando como consecuencia la obtención de un plan global.

Las técnicas que combinan planificación con CBR reciben el nombre de *planificación basada en casos* (CBP), y se basan en crear casos que representan o bien planes completos o bien planes parciales (en el segundo caso dichos planes deben ir ensamblándose, para lo cual se debe incluir algún mecanismo de control que se encargue de ello).

La CBP se ha aplicado con éxito en numerosos ámbitos, como el diagnóstico de enfermedades (ASP-II [2]), la planificación navegacional (SINS [29]), o el diseño de software (DÉJÀ VU [31]), entre otros.

1.3. Darmok, un Sistema CBP para Juegos de Estrategia

Los sistemas Darmok, *Darmok* [25] y *Darmok 2* [23] son sistemas CBP orientados a juegos de estrategia en tiempo real (aunque en teoría Darmok 2 puede jugar a cualquier juego de competición entre adversarios). Darmok[2] implementa una filosofía de *aprendizaje mediante ejemplos*: un jugador experto muestra al sistema cómo jugar, y éste, automáticamente, aprende a partir de las demostraciones. Las demostraciones se almacenan en la base de casos en forma de planes, de modo que, durante su ejecución, Darmok emplea dichos casos para construir un plan jerárquico (al estilo de las HTNs) con el que ganar la partida.

Algunas de las mayores limitaciones de la planificación clásica, incluyendo los sistemas CBP, es que asumen tanto la presencia de un mundo estático como que disponen de un tiempo ilimitado para trazar sus planes. El hecho de que una gran parte de los dominios del mundo real tenga esta doble restricción ha conllevado estudios acerca de cómo darle una solución. En particular, los juegos de estrategia de tiempo real son un claro ejemplo de este doble requerimiento, el cual debe ser tomado en consideración.

Darmok implementa una versión modificada del ciclo CBR, a saber, el ciclo *CBR en línea* (OLCBR).

El ciclo OLCBR (figura 1.2) [25] es una versión adaptada del ciclo CBR para dominios de planificación de estrategia en tiempo real. El ciclo tradicional presupone que un problema puede resolverse con una sola vuelta al ciclo. En CBP, sin embargo, es posible que el problema original se tenga que descomponer en subproblemas, para cada uno de los cuales el ciclo de CBR debería volver a aplicarse. En planificación clásica, además, normalmente se supone que el plan, una vez construido, tendrá éxito. En dominios de estrategia en tiempo real, sin embargo, puede no darse el caso, ya que en muchas ocasiones el conocimiento que se tiene del mundo no es lo suficientemente preciso, o simplemente, al ser el entorno dinámico, el plan no puede completarse. Asegurarse de que el plan se ejecuta correctamente es pues un aspecto importante en este tipo de dominios.

El ciclo OLCBR incluye dos fases adicionales para hacer frente a la planificación basada en casos en dominios de estrategia en tiempo real:

- Expansión: encargada de tomar la solución actual (plan), y analizar si en él hay subproblemas actualmente abiertos. Si los encuentra, los envía al comienzo del ciclo CBR (fase de recuperación), para resolverlos. Además, el módulo de expansión es el encargado de realizar *adaptación retrasada* de planes en caso de que, cuando vayan a ejecutarse, el estado del mundo haya cambiado lo suficiente como para que puedan no ser aplicables.

- Ejecución: encargada de ejecutar el plan y actualizar su estado de ejecución. Si algún subplan falla, el estado de ejecución del plan global es actualizado, para que el módulo de expansión pueda encontrar una solución alternativa.

El funcionamiento de Darmok puede dividirse en dos fases:

[2]De ahora en adelante, salvo que se especifique lo contrario, usaremos *Darmok* para referirnos a *Darmok 2*, su versión más avanzada.

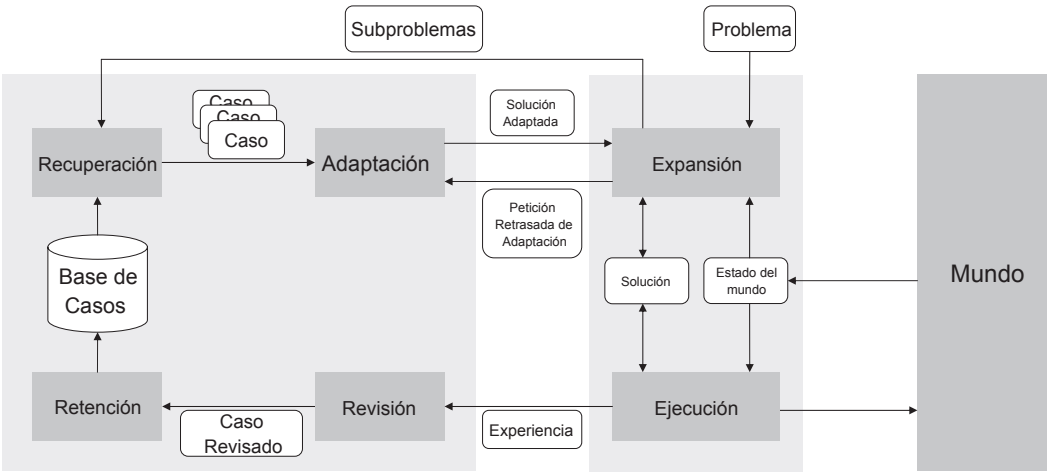

Figura 1.2: El ciclo OLCBP

- Fase de aprendizaje: donde el experto juega al videojuego, generándose *trazas* que describen de forma precisa todos los eventos acaecidos durante la partida. Estas trazas son almacenadas en forma de planes (casos del sistema), que son empleados posteriormente durante la fase de ejecución.
- Fase de ejecución: donde se aplica el ciclo OLCBP para mantener un plan global (compuesto a su vez de subplanes) que gestiona la partida.

1.3.1. Representación de Planes

En Darmok, la pieza básica constituyente de un plan es el *snippet* , el cual representa una serie de instrucciones (acciones) a ejecutar. Un snippet consta de:

- Un conjunto de *precondiciones*, las cuales deben ser satisfechas para que el plan pueda ejecutarse (por ejemplo, antes de poder construir un tanque, se deben tener los recursos suficientes para ello).
- Un conjunto de *condiciones de vida*, las cuales deben mantenerse ciertas durante la ejecución del plan. Si no, el plan se considera fallido.
- Un conjunto de *condiciones de éxito*, las cuales determinan cuándo el plan ha tenido éxito. Obsérvese que las condiciones de éxito no son lo mismo que las postcondiciones. A diferencia de la planificación clásica, no se pueden especificar postcondiciones, ya que no está garantizado que un plan vaya a tener éxito.
- Un plan.

Mientras que en Darmok 2 el plan se almacena como una red de petri jerárquica, en Darmok el plan está compuesto de construcciones de tipo *secuencia*, *paralelo*, *acción* y *subobjetivo*. En cualquier caso, ambas son representaciones jerárquicas que permiten descomponer un plan global en subplanes asociados a cada subobjetivo.

Para poder usar Darmok se debe proporcionar información específica del domino del juego (definidas en un *fichero de dominio* escrito en el lenguaje de *MakeMEPlayME* [17], un framework para conectar sistemas de IA a juegos de estrategia en tiempo real), incluyendo:

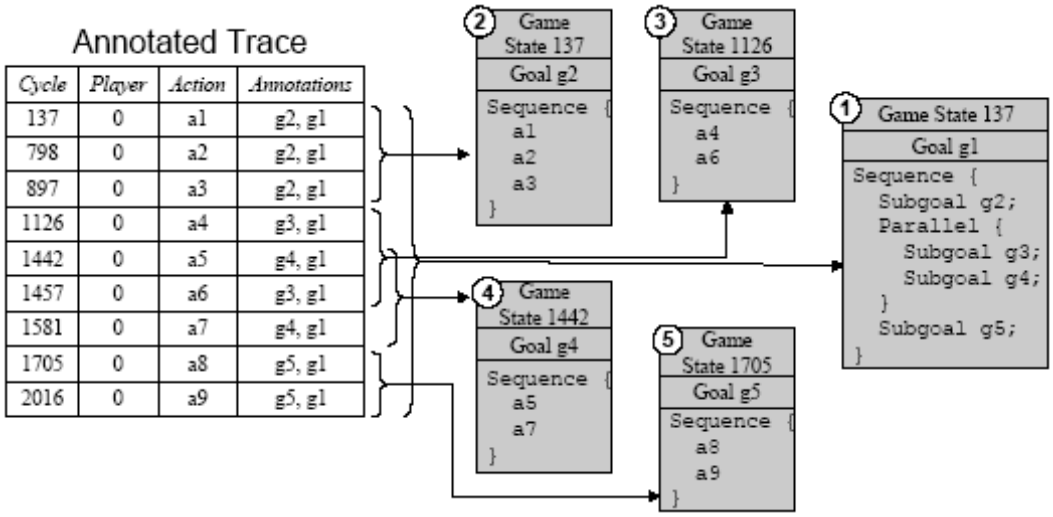

Figura 1.3: Extracción de casos en Darmok a partir de una traza

- *Acciones* básicas que se pueden ejecutar en el dominio, como por ejemplo *Atacar* o *Moverse*.
- *Sensores*, que permiten obtener información acerca del estado del mundo.
- *Objetivos*, los cuales pueden estar organizados en una jerarquía.

1.3.2. Adquisición de Planes

En la fase de adquisición de planes se obtienen los planes que almacenará la base de casos con la cual Darmok podrá construir un plan para el desarrollo de la partida. En Darmok, los casos de la base de casos están formados por snippets y por *episodios* asociados a los snippets. Mientras que el snippet representa un plan concreto, el episodio almacena el resultado de aplicar dicho plan en un estado del mundo concreto para satisfacer un objetivo concreto.

El problema de la adquisición de planes a partir de la traza de una partida es la imposibilidad de saber qué objetivos perseguía el jugador con cada una de las acciones de la traza.

En Darmok (que no Darmok 2), este problema se resolvía mediante la *anotación manual de la traza*, en la cual un experto recorría la traza de acciones e indicaba, para cada una de ellas, el objetivo que perseguía (el objetivo es seleccionado de entre el listado de objetivos especificados en el dominio). Una vez que se tiene, para cada acción, qué objetivo se perseguía con ella, se lleva a cabo un análisis temporal de la traza (ver figura 1.3), el cual permite extraer los casos (snippets y episodios asociados) con los que rellenar la base de casos.

En Darmok 2 la adquisición de casos se realiza de forma automática, a través de una *matriz de objetivos* y un *grafo de dependencias* [23, 34]. Gracias a la matriz de objetivos, se construyen planes básicos para cada objetivo, los cuales son posteriormente simplificados y ordenados en una jerarquía de subojetivos mediante el uso del grafo de dependencias.

En cualquier caso, ya se trate de Darmok o Darmok 2, el resultado de la fase de aprendizaje son casos en forma de snippets y episodios que se usan en la posterior fase de ejecución.

1.3.3. Recuperación de Planes

El ciclo OLCBP que implementa Darmok descompone el objetivo inicial en subobjetivos, y estos a su vez en subsubobjetivos, de manera recursiva. En cada caso, para resolver cada uno de dichos objetivos, Darmok debe recurrir a su base de casos y encontrar qué planes (snippets) son más adecuados para cada circunstancia.

La base de casos de Darmok contiene casos de la forma $c = \langle p, e \rangle$, donde p representa el snippet, y donde $e = \langle G, S, O \rangle$ representa el episodio (siendo G el objetivo perseguido, S el estado del mundo y O un valor numérico entre 0 y 1 indicando la medida del éxito de la aplicación del plan de p para la consecución de G en el estado S).

La fase de recuperación de Darmok tiene como objetivo, dado un objetivo a resolver y un estado del mundo, encontrar el plan que, previsiblemente, mejor lo resuelva. Para ello se aplica la medida de *relevancia de episodio*, que, dado un objetivo y un estado del mundo define cómo de importante es un episodio respecto a dicho estado y a dicho objetivo:

$$ER(e, S, G) = \alpha GS(e.G, G) + (1 - \alpha)SS(e.S, S)$$

Donde GS es una medida de distancia entre objetivos (con valores entre 0 y 1), SS es una medida de distancia entre estados del mundo (con valores entre 0 y 1), y donde α es una constante entre 0 y 1 que indica la importancia de cada una de las medidas de distancia.

Por último, para predecir el rendimiento de un snippet, se toman los k episodios más relevantes asociados a dicho snippet, y se obtiene una media ponderada del éxito ($e.O$) de los k episodios relevantes. El resultado de la fase de recuperación es el snippet de mayor rendimiento.

1.3.4. Fase de Expansión y Ejecución En Linea

La fase de ejecución de Darmok está gestionada por tres módulos independientes que interaccionan entre sí: el *módulo de expansión*, el *módulo de ejecución*, y el *módulo de adaptación*. Entre ellos van construyendo un plan en forma de árbol que van desarrollando durante el juego. Dicho árbol contiene dos tipos de nodos, *nodos snippet* y *nodos objetivo*.

Inicialmente, el plan contiene un sólo nodo, el objetivo *GanarJuego*. El módulo de expansión pide al módulo de recuperación de casos un snippet para satisfacer dicho objetivo. Dicho snippet podría tener a su vez subobjetivos, para los cuales el módulo de expansión debería volver a pedir al módulo de recuperación un snippet adecuado, y así sucesivamente. En general, cuando el módulo de expansión encuentra en el plan nodos objetivo que están *abiertos* (es decir, no tienen ningún snippet asociado o el snippet que se les asoció falló) y *preparados* (es decir, no hay ningún snippet que se tenga que ejecutar antes), consulta al módulo de recuperación para obtener un snippet para dicho objetivo.

Mientras tanto, de forma paralela, el módulo de ejecución se encarga de comenzar la ejecución de cada snippet tan pronto como compruebe que sus precondiciones se satisfacen. Si el estado del mundo ha cambiado en gran medida respecto a cuando el snippet fue recuperado de la base de casos, éste es enviado al módulo de adaptación para intentar adecuarlo al estado actual. Si

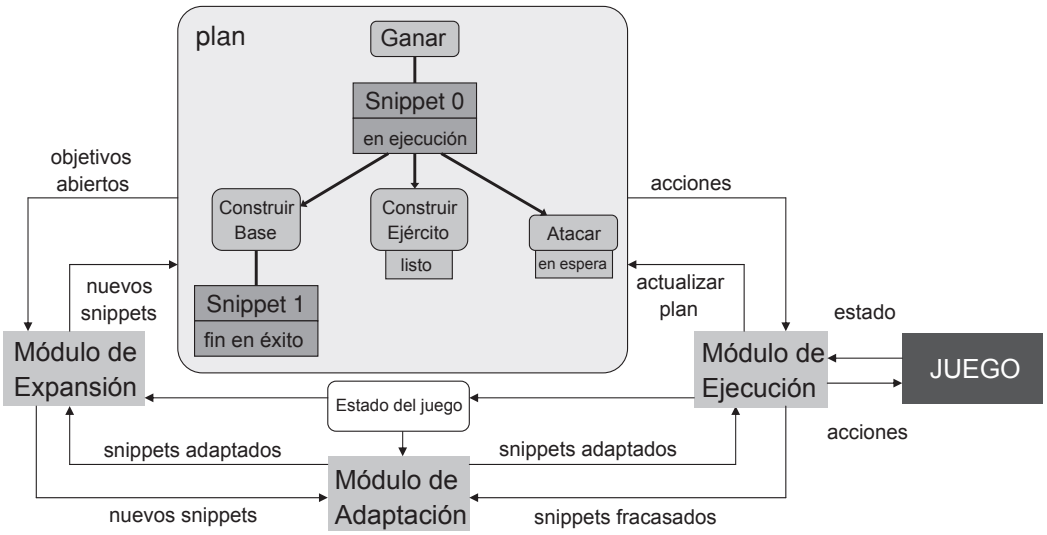

Figura 1.4: Fase de ejecución en Darmok

algún snippet en ejecución contiene acciones básicas que pueden ser ejecutadas (sus precondiciones se satisfacen), se envían al motor de IA subyacente para que sean ejecutadas. Si las precondiciones no se satisfacen, el snippet es enviado al módulo de adaptación para repararlo. Además, el módulo de ejecución comprueba periódicamente las condiciones de vida y de éxito de cada snippet (y acción básica), de modo que, cuando el snippet (o acción) se da por finalizado, se actualiza el estado del snippet padre: si ha finalizado con éxito, el snippet padre puede continuar su ejecución; si ha fracasado, el snippet se marca como fallido, y su objetivo asociado se marca como abierto nuevamente.

La figura 1.4 muestra la ejecución de un plan. El objetivo global, *Ganar-Juego*, ha decidido resolverse mediante la ejecución del snippet 0. El plan de dicho snippet contiene tres subobjetivos, *Construir Base*, *Construir Ejército* y *Atacar*. El objetivo *Construir Base* ha sido cumplido, ya que el plan que se recuperó para resolverlo, snippet 1, ha finalizado con éxito. El objetivo *Construir Ejército* está listo y a la espera de que el módulo de expansión le proporcione un snippet adecuado. Mientras tanto, el último objetivo, *Atacar*, está en espera de que se cumpla el objetivo *Construir Ejército*.

1.3.5. Adaptación

Como en todo sistema CBR, en Darmok es necesario llevar a cabo una fase de adaptación de los casos almacenados en la base de casos. Cuando se va aplicar un plan, es posible que éste no sea aplicable al estado actual del mundo, motivo por el cual debería ser revisado.

En Darmok se dan dos tipos de adaptación, la *adaptación de parámetros* y la *adaptación estructural*.

La adaptación de parámetros, más sencilla, tiene como objetivo adaptar los parámetros de las acciones básicas que componen los planes de la base de casos. Las acciones básicas, como *Moverse* o *Atacar*, tienen parámetros (por ejemplo, *moverse a la posición (2,3)* o *atacar a la unidad 34*). Cuando se recupera un plan de la base de casos, es probable que los parámetros de sus acciones básicas no sean aplicables al estado actual del juego, así que Darmok lleva a cabo un paso de adaptación previo antes de aplicarlas.

La adaptación estructural es sensiblemente más compleja que la adaptación de parámetros, y permite adaptar la estructura (a nivel de acciones) del plan original.

La adaptación estructural permite tanto eliminar del plan acciones innecesarias como añadirle nuevas acciones que permitan hacer ciertas precondiciones no satisfechas inicialmente. Para ello se hace uso del grafo de dependencias, el cual muestra las dependencias existentes entre las distintas acciones que lo conforman, indicando cuándo el éxito de alguna de ellas (evaluable mediante las condiciones de éxito) contribuye a la satisfacción de las precondiciones de otra. La eliminación de acciones innecesarias hace uso directo del grafo del plan, y permite eliminar aquellas acciones que no tienen una dependencia directa en la consecución del objetivo del plan. Este tipo de situaciones se dan con frecuencia durante el juego, ya que es de esperar que haya acciones del plan original cuyas condiciones de éxito sean ciertas en la situación actual del juego (distinta de cuando el plan fue construido), y que por tanto no tengan por qué ejecutarse. Por otro lado, es frecuente que, cuando un plan vaya a ejecutarse, sus precondiciones no sean ciertas en la situación actual. En ese caso se lleva a cabo un tipo de adaptación estructural que permite insertar nuevos subplanes al plan original, con la idea de que la realización de cada uno los subplanes satisfaga alguna de las precondiciones no satisfechas.

Capítulo 2

Árboles de Comportamiento

Tradicionalmente, los sistemas de decisión de bajo nivel que controlan los agentes (personajes) de videojuegos han hecho uso de algún tipo de variante de máquina de estados finitos (FSMs), siendo las más conocidas las máquinas de estados finitos jerárquicas (HFSMs). En contextos donde la inteligencia de los agentes es demasiado compleja, desafortunadamente, las FSMs se muestran en ocasiones insuficientes, de modo que otro tipo de enfoques han sido adoptados, destacando los lenguajes de *scripting*, que permiten un grado de control mucho mayor sobre el comportamiento de la entidad.

En los últimos años, una nueva técnica conocida como *árboles de comportamiento* se ha convertido en una alternativa bastante popular en la creación de las IAs de personajes de videojuegos. Los árboles de comportamiento han sido usados en videojuegos de última generación tales como Halo, Spore o GTA: Chinatown Wars.

Los árboles de comportamiento pueden verse como una evolución natural de las HFSMs que favorece la reutilización de comportamientos reemplazando las transiciones explícitas de las FSMs con mecanismos procedurales que permiten calcular cuál el siguiente estado del comportamiento del agente.

Aunque los árboles de comportamiento inicialmente nacieron como una herramienta dirigida a programadores, con el paso del tiempo los diseñadores profesionales de videojuegos han comenzado a usarlos de manera constante para la creación de los comportamientos de los personajes [9, 12]. Quizás uno de los puntos claves de los árboles de comportamiento es que, tal y como ocurre con las FSMs, es fácil concebir entornos de desarrollo integrado que permitan crear y editar comportamientos a través de una interfaz gráfica, lo cual favorece y acelera en gran medida el proceso de desarrollo. Otras técnicas, como los *scripts*, no son propicias a ello, lo cual las hacen más inefectivas.

Los árboles de comportamiento son estructuras jerárquicas de tipo árbol cuya principal componente es la *tarea*. En un árbol de comportamiento, un nodo del árbol representa una tarea, y una tarea representa un comportamiento. Mientras que las hojas del árbol son tareas de *bajo nivel* que se ejecutan en el entorno virtual del juego, los nodos intermedios representan tareas compuestas (tareas con uno o varios hijos en la estructura del árbol), las cuales determinan de qué modo evoluciona la ejecución del árbol (y en última instancia, en consecuencia, qué tareas de bajo nivel se ejecutarán). En cualquier caso, el nodo que encarna un comportamiento determinado puede o bien *fallar* o bien *tener éxito* en su ejecución. Dependiendo del resultado, la tarea (comportamiento) padre toma el control de la ejecución del árbol y reacciona acorde a su semántica.

Es justamente el hecho de que todos los comportamientos (tareas o nodos del árbol) comparten la misma interfaz de funcionamiento lo que hace que los árboles de comportamiento sean una herramienta potente y a su vez sencilla. Esto permite que se puedan construir comportamientos jerárquicos con gran facilidad, partiendo de comportamientos simples para agruparlos posteriormente en comportamientos más complejos. Como en la mayoría de los casos los comportamientos son auto-contenidos, además, los nuevos comportamientos no deben preocuparse acerca de los detalles del funcionamiento de cada subcomportamiento, pudiendo reutilizarlos de manera transparente.

Una de las mayores ventajas de los árboles de comportamiento es su capacidad para determinar cuándo se ejecuta un determinado comportamiento sin necesidad de definir, como ocurre con las FSMs, condiciones explícitas para cada tipo de transición. En lugar de ello, los árboles de comportamiento hacen uso de *guardas*. Cada tarea del árbol puede ser etiquetada con una guarda que debe ser evaluada como cierta para que el comportamiento se active. A este respecto, los árboles de comportamiento incluye la tarea compuesta conocida como *lista estática de prioridad*, la cual permite al diseñador etiquetar a cada uno de sus hijos con una guarda. Cuando la lista se ejecuta, las guardas de sus hijos se evalúan, de modo que sólo se ejecuta el primer hijo cuya guarda haya sido evaluada como cierta. Una variante de este tipo de tarea es la *lista dinámica de prioridad*, la cual evalúa continuamente las guardas de sus hijos, pudiendo abortar la ejecución del hijo actualmente activo si la guarda de algún otro de mayor prioridad se evalúa como cierta (en cuyo caso el hijo de mayor prioridad comienza su ejecución).

Gracias a las guardas, los árboles de comportamiento pueden concebirse como estructuras orientadas por objetivos que representan cómo un objetivo de alto nivel puede descomponerse en objetivos de bajo nivel. En este sentido, los árboles de comportamiento se asemejan a las redes de tareas jerárquicas (HTNs) que se usan en planificación automática, si bien su propósito es totalmente diferente. Mientras que las HTNs se usan para generar planes, los árboles de comportamiento se usan para almacenar planes codificados manualmente. Los árboles de comportamiento pueden entenderse como árboles de tipo *and-or* que almacenan planes gracias a los cuales una entidad puede conseguir sus objetivos.

En este capítulo describimos de forma detallada los árboles de comportamiento. En primer lugar comentamos su mecánica de funcionamiento. Posteriormente hablamos del contexto de ejecución, que juega un papel especialmente relevante dentro del funcionamiento de los árboles de comportamiento. Finalmente describimos las tareas que comúnmente se utilizan para su construcción y analizamos algunos ejemplos de árboles.

2.1. Mecánica de Funcionamiento

La mecánica de funcionamiento de los árboles de comportamiento se basa en el hecho de que todas las tareas (nodos) del árbol comparten una misma interfaz de operación. El modo en que un nodo interacciona con sus hijos es independiente del tipo de cada hijo, es decir, del comportamiento que representa, y sólo depende de la semántica del padre.

Inicialmente, un ejecutor externo pide al nodo raíz del árbol, es decir, al comportamiento que se quiere ejecutar, que comience su ejecución. El nodo

```
1 public class BTNode {
2     private Vector<BTNode> children;
3
4     public void spawn(Context context);
5     public Status update();
6     public void abort();
7 }
```

Figura 2.1: Estructura de un nodo de un árbol de comportamiento

raíz, en base a su semántica (es decir, a su tipo), comenzará la ejecución de uno o varios de sus hijos, los cuales, dependiendo a su vez de su tipo, comenzarán asimismo la ejecución de uno o varios de sus hijos, y así sucesivamente. El proceso finaliza cuando la ejecución alcanza las hojas del árbol (tareas de bajo nivel), las cuales llevan a cabo acciones en el entorno virtual del videojuego. Dependiendo del estado de finalización de las tareas de bajo nivel (*éxito* o *fracaso*), sus respectivos padres actuarán de una manera u otra, bien iniciando la ejecución de nuevos hijos, bien considerando finalizada su ejecución (en éxito o fracaso) y propagando el control de la ejecución de vuelta a sus padres, de forma recursiva.

A nivel conceptual, una tarea o nodo de un árbol de comportamiento muestra la interfaz de la figura 2.1.

El método *spawn()* es el que desencadena la ejecución del nodo. Para tareas de bajo nivel, generalmente se encarga de enviar órdenes a las entidades del juego o bien analizar el estado de algunas variables de interés del mundo. En el caso de tareas intermedias, nodos compuestos, *spawn()* se encarga de, acorde a la semántica del nodo, lanzar recursivamente la ejecución de uno o varios de sus hijos (llamando para ello a sus respectivos métodos *spawn()*).

Una vez que se ha comenzado la ejecución de una tarea, el método *update()* se encarga de actualizar su estado de ejecución. Para tareas de bajo nivel, el método *update()* analiza el estado de terminación de la tarea que se está realizando en el videojuego, y dependiendo de él, devuelve un código (*Status*) de éxito (*SUCCESS*), fracaso (*FAILURE*) o no-terminación (*RUNNING*). Para tareas intermedias, el método *update()* llama de manera recursiva al método *update()* de sus hijos activos, analiza sus estados de terminación, y dependiendo de ellos devuelve uno u otro estado de terminación.

El método *abort()* se emplea en situaciones en las que se debe terminar de manera abrupta la ejecución de una tarea.

La estructura descrita para las tareas de los árboles de comportamiento es meramente conceptual, con la finalidad de describir de manera formal cómo procede su ejecución. A la hora de implementarlos, sin embargo, se pueden seguir diversas filosofías.

2.2. Contexto de Ejecución

Todos los nodos de un árbol de comportamiento disponen de un *contexto de ejecución*, el cual es compartido por todos ellos. El contexto de ejecución, o simplemente *contexto*, actúa como una pizarra que pueden usar los nodos para escribir y leer variables. El contexto no es más que un repertorio de varia-

```
1  public class Context {
2      private Hashtable<String,Object> variables;
3
4      public void setVariable(String name, Object value){
5          variables.put(name,value);
6      }
7
8      public Object getVariable(String name){
9          return variables.get(name);
10     }
11 }
```

Figura 2.2: El contexto de ejecución

bles, donde cada variable es un par de la forma (*nombre de variable*, *valor*). El contexto representa la porción del estado del juego que es accesible a la entidad controlada por el árbol. Toda la información que la entidad necesite del mundo sólo puede ser accedida a través del contexto, ya que los árboles de comportamiento no tienen otro modo de comunicarse con el exterior. La figura 2.2 muestra cómo se define el contexto de ejecución a nivel de pseudocódigo.

En general, todo contexto dispone de al menos dos variables:

- *?this*: que referencia al agente que está siendo controlado por el árbol. A través de esta variable se puede acceder a las propiedades de dicho agente.
- *?world*: que referencia al mundo donde el juego tiene lugar.

No obstante, el contexto tiene una utilidad mucho más amplia, ya que es el mecanismo a través del cual los nodos del árbol pueden comunicarse entre sí mediante el intercambio de variables. Al actuar como una pizarra de lectura y escritura, las tareas (generalmente las hojas del árbol) pueden escribir variables en el contexto para que posteriormente puedan ser usadas por otros nodos. Por ejemplo, un nodo podría escribir un variable de nombre *MiVariable* en el contexto, la cual podría ser leída posteriormente por otro nodo, accediendo por su nombre, *MiVariable*. Se puede establecer así una semántica de comunicación entre tareas.

A una tarea se le pueden asociar una serie de parámetros de entrada necesarios para su correcta ejecución. Cuando ésta comienza, a cada parámetro se le asocia un valor, bien literal, bien leído del contexto de ejecución.

Por normal general, todos los nodos de un árbol comparten el mismo contexto, ya que éste se suele pasar de los nodos padres a los nodos hijos (a través del método *spawn()* del pseudocódigo de la figura 2.1). Se tiene entonces que el contexto inicial es el de la tarea raíz del árbol, que es transferido a sus hijos, los cuales harán lo propio con sus hijos, y así sucesivamente. En algunos escenarios es deseable, a pesar de todo, que algunos subárboles dispongan de su propio área de almacenamiento de datos privada, para que cualquier tipo de modificación no afecte al contexto del resto del árbol. En ese caso se rompe la regla de transferencia del contexto de padres a hijos, pasándoles en su lugar un nuevo contexto. Para emular un paso tradicional de contexto y que los hijos tengan la sensación de trabajar con el contexto original, el nuevo contexto

puede contener una copia de las variables del contexto padre. Así, por un lado, el subárbol interacciona con el nuevo contexto como lo habría hecho con el contexto original, pero por otro no lo modifica.

2.3. Tipos de Tareas

Definida la semántica de funcionamiento de los árboles de comportamiento, hay que establecer un conjunto de nodos suficientemente amplio y potente como para que se puedan implementar comportamientos complejos. Debe tenerse en cuenta que el objetivo de los árboles de comportamiento es controlar de forma precisa y realista a personajes no jugadores en videojuegos, y que para ello se requiere contemplar en general multitud de situaciones.

En la terminología de los árboles de comportamiento normalmente se habla de tres tipos de tareas: tareas compuestas, decoradores y tareas de bajo nivel (acciones y condiciones). Las tareas compuestas constan de uno o varios hijos, y su estado de ejecución depende del estado de ejecución de éstos. Los decoradores, basados en el patrón software del mismo nombre, tienen un único hijo, cuyo comportamiento alteran de cierta manera. Por último, las tareas de bajo nivel se dividen en dos subgrupos: las acciones, que representan tareas que alteran el estado del videojuego, y que se ejecutan contra éste, y las condiciones, que simplemente consultan el estado del mundo para ver si se satisface una determinada condición (también deben ejecutarse contra el videojuego).

2.3.1. Tareas de Bajo Nivel

Las tareas de bajo nivel son las más simples conceptualmente dentro de los árboles de comportamiento. Éstas forman las hojas de los árboles, y se dividen en dos tipos, *acciones* y *condiciones*.

Las tareas de bajo nivel se comunican directamente con el videojuego (preferentemente a través de algún tipo de API) para llevar a cabo actividades de consulta sobre el estado del mundo o bien para modificarlo.

En este sentido, las **acciones** son tareas de bajo nivel que modifican el estado del mundo (y que también pueden leerlo). Son acciones tareas como ordenar a la entidad controlada moverse a una determinada posición, o bien ordenarle desenfundar un arma. Si la orden realizada tiene éxito, la acción finaliza en éxito; de otro modo finaliza en fracaso. Por otro lado, las **condiciones** son tareas de bajo nivel que comprueban que se cumple cierta propiedad en el estado del mundo (a diferencia de las acciones, las condiciones no alteran el estado del mundo). Las condiciones se muestran extremadamente útiles cuando se quiere realizar acciones sólo bajo ciertas circunstancias. En ese caso, las acciones se ven guardadas por una condición, garantizando que, si la acción se ejecuta, es porque la condición se cumple (por ejemplo, la acción *atacar al enemigo con granada* sólo debería ejecutarse si el personaje dispusiera de una granada).

2.3.2. Tareas Compuestas

Una tarea compuesta es una tarea que dispone de uno o varios hijos, y cuya ejecución depende del estado de ejecución de éstos. Cuando una tarea compuesta comienza a ejecutarse, lanza a ejecución uno o varios de sus hijos,

dependiendo de su semántica, y monitoriza la evolución de éstos. Una vez finalizados los hijos, dependiendo de sus estados de ejecución (éxito o fracaso), la tarea compuesta bien puede lanzar a ejecución a otros hijos, o bien decidir finalizar ella misma en éxito o en fracaso.

Los dos tipos más útiles de tareas compuestas son la *secuencia* y el *selector*.

El nodo de tipo **secuencia** contiene uno o varios hijos, que ejecuta secuencialmente. Cuando el nodo *secuencia* es ejecutado, lanza al primero de sus hijos. Si éste finaliza en éxito, la *secuencia* continúa con el siguiente de sus hijos. Si por contra falla, la *secuencia* se considera fallida. En general, el nodo *secuencia* tiene éxito si todos sus hijos logran ejecutarse (secuencialmente) con éxito. Si alguno de ellos falla, la *secuencia* se considera fallida. Una *secuencia* representa una serie de tareas que deben ser realizadas para cumplir un objetivo determinado. El pseudocódigo de la figura 2.3 muestra una implementación del nodo *secuencia* acorde al prototipo de la figura 2.1.

La figura 2.4 muestra un árbol de comportamiento simple con un nodo *secuencia* como raíz y con tres tareas de bajo nivel. Este árbol recoge el comportamiento de un personaje que huye ante la presencia de un enemigo. En efecto, cuando el árbol comienza su ejecución en su raíz, el nodo *secuencia*, éste comienza a ejecutar secuencialmente a sus hijos. El primer hijo es una condición que comprueba si hay algún enemigo a la vista. De no haberlo, la condición falla, y como consecuencia falla la *secuencia*, no ejecutándose así las otras dos tareas de bajo nivel. No obstante, si la condición es cierta, la *secuencia* pasa a ejecutar el siguiente hijo, la acción *Darse la vuelta*. Si ésta tiene éxito, finalmente se ejecuta la acción *Huir*. El árbol representa el comportamiento *Huir ante el peligro*, para cuyo éxito es necesario que se ejecuten exitosamente las tres tareas de bajo nivel que lo componen.

El nodo de tipo **selector** muestra un comportamiento parecido, ya que éste ejecuta secuencialmente a sus hijos. Sin embargo, tan pronto como uno de ellos finalice con éxito, el *selector* también finaliza con éxito. Si alguno de los hijos del *selector* acaban en fracaso, el *selector* no finaliza, sino que sigue ejecutando al siguiente de sus hijos. El *selector* sólo se considera fallido cuando todos sus hijos fallan. El *selector* puede verse como una tarea que intenta realizar una determinada acción de diversas maneras. Si una de ellas falla, prueba la siguiente, hasta que se acaban todas las opciones disponibles.

El árbol de la figura 2.5 muestra un árbol de comportamiento simple con un nodo *selector* y tres acciones de bajo nivel. El árbol representa el comportamiento *Hacer feliz a madre*, y su propósito es hacer feliz a una madre triste mediante una serie de alternativas, a cada cual más ruin. Cuando el árbol comienza su ejecución en el nodo raíz, el *selector*, éste expande al primero de sus hijos e intenta ejecutarlo. Si la acción *Decir "te quiero"* tiene éxito, la madre es ya feliz, y el comportamiento (nodo *selector*) finaliza. Sin embargo, si fracasa, se intenta la siguiente opción para hacer a la madre feliz, ejecutándose la acción *Hacer un postre*. Si tiene éxito, el árbol finaliza en éxito, pero si fracasa, se intenta la última de las opciones, *Hacer un regalo*. En efecto, el árbol intenta satisfacer el objetivo inicial mediante tres alternativas distintas, y mientras alguna de ellas tenga éxito, el comportamiento se considerará exitoso.

Los nodos *secuencia* y *selector* son las tareas compuestas más comunes cuando se diseñan árboles de comportamiento. Éstas permiten diseñar todo tipo de comportamientos, desde los más simples (como los de las figuras 2.4 y 2.5) a otros muchos más complejos, como el de la figura 2.6.

```
public class Sequence extends BTNode {
    /* Indice del hijo activo. */
    private int activeChild = -1;

    public void spawn(Context context){
        /* Comenzar la ejecucion del primer hijo */
        activeChild = 0;
        children[activeChild].spawn(context);
    }

    public Status update(){
        /*
         * Actualizar el estado de ejecucion del hijo
         * activo, y avanzar la ejecucion del nodo
         * selector acorde a ello.
         */
        Status childStatus;
        childStatus = children[activeChild].update();

        if(childStatus == RUNNING || childStatus == FAILURE){
            return childStatus;
        }
        else{
            /*
             * El hijo ha finalizado con exito. Si es
             * el ultimo, se devuelve exito. Si no, se
             * comienza la ejecucion del siguiente hijo.
             */
            if(activeChild == children.size()){
                return childStatus;
            }
            else{
                activeChild++;
                children[activeChild].spawn();
                return RUNNING;
            }
        }
    }

    public void abort(){
        /* Abortar la ejecucion del hijo activo. */
        if(activeChild != -1 && activeChild != children.size()){
            children[activeChild].abort();
        }
    }
}
```

Figura 2.3: Pseudo-implementación del nodo *secuencia*

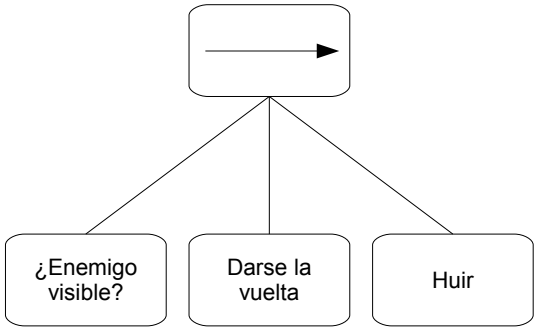

Figura 2.4: Un árbol de comportamiento simple con un nodo *secuencia*

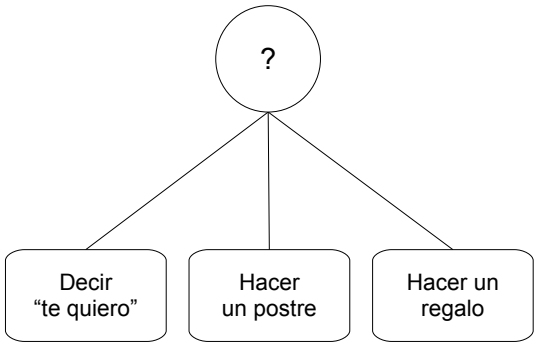

Figura 2.5: Un árbol de comportamiento simple con un nodo *selector*

El árbol de la figura 2.6 conceptualiza el comportamiento *Entrar en habitación*. El comportamiento recoge dos casos (hijos *secuencia* del nodo *selector* raíz). Si cuando se va a entrar a una habitación la puerta está abierta (primera *secuencia* hija de nodo raíz), entonces el personaje entra simplemente en la habitación. Sin embargo, si no lo está (segunda *secuencia* hija del nodo raíz), el personaje debe intentar abrirla. Para ello, en primer lugar se acerca a la puerta. Luego intenta abrirla de dos modos: si la puerta está cerrada con llave, se intenta abrir con la llave. Obsérvese que si la acción *Abrir puerta con llave* tiene éxito, no se intentará el siguiente modo de abrir la puerta. Por contra, si la puerta no está cerrada con llave, o si la llave no funciona correctamente (*Abrir puerta con llave* falla), se intenta otra alternativa para abrir la puerta, que es romperla embistiéndola. Si tras embestir contra ella la puerta se percibe como abierta, se considerará que el intento de derribarla ha tenido éxito. En cualquier caso, si el subárbol que intenta abrir la puerta (bien con la llave bien derribándola) tiene éxito, el personaje entra en la habitación. Si la puerta no ha podido ser abierta, el comportamiento finalmente se considera fracasado.

Existen variantes de los nodos *secuencia* y *selector* que, en lugar de ejecutar sus hijos de forma secuencial, primero los barajan de forma aleatoria, y luego los ejecutan en el orden establecido. Estas variantes se emplean en situaciones en las que se quiera conferir un mayor realismo al comportamiento, ya que elimina el componente de predictibilidad asociado a una ejecución secuencial estática.

Otros tipos de tareas compuestas más sofisticadas permiten diseñar comportamientos más complejos y reactivos.

El nodo **paralelo**, por ejemplo, permite ejecutar de forma concurrente a todos sus hijos. Los nodos *paralelos* son útiles en situaciones en las que, por ejemplo, se quiere asegurar que una condición se mantiene válida al mismo

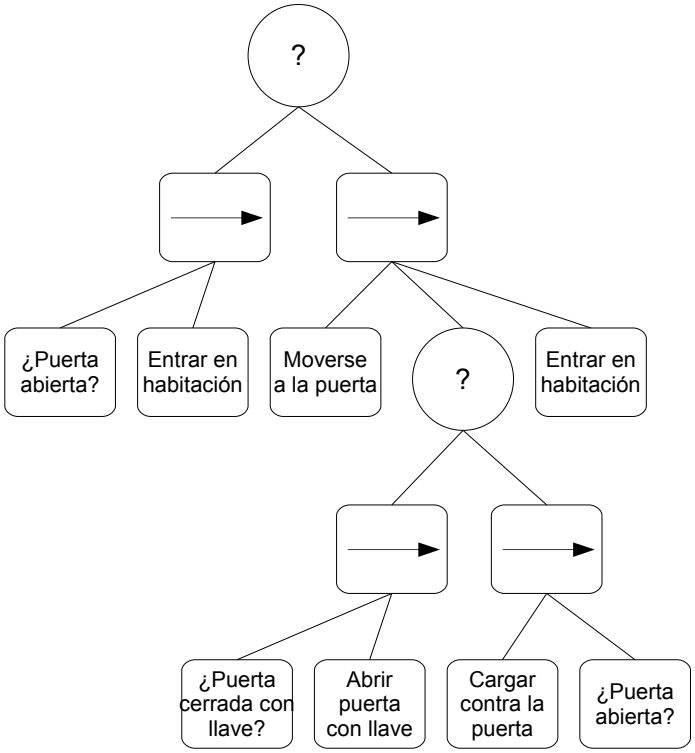

Figura 2.6: Un árbol de comportamiento para el comportamiento *Entrar en habitación*

tiempo que se ejecuta otro comportamiento.

El concepto de *guarda* toma forma en las **listas estáticas de prioridad** y en las **listas dinámicas de prioridad**. Ambas permiten etiquetar a sus hijos con una condición (guarda), que debe ser evaluada como cierta antes de que los hijos comiencen a ejecutarse. La *lista estática de prioridad* evalúa inicialmente las guardas de sus hijos, y comienza a ejecutar aquél de mayor prioridad (situado más a la izquierda en el lista de sus hijos) cuya guarda sea cierta.

Aunque el comportamiento de la *lista dinámica de prioridad* es similar, ésta representa una estructura más reactiva. *La lista dinámica de prioridad* comienza ejecutando al hijo de mayor prioridad cuya guarda es evaluada como cierta. A partir de entonces, la lista reevalúa constantemente las guardas, de modo que si alguna guarda de mayor prioridad se hace cierta, se cancela al hijo actualmente en ejecución, y comienza la ejecución del nuevo hijo.

2.3.3. Decoradores

Un nodo decorador es una tarea que modifica el comportamiento de otra tarea. Los decoradores están basados en el patrón software del mismo nombre. En orientación a objetos, el patrón decorador se refiere a una clase que encapsula a otra. Si el decorador tiene la misma interfaz que la de la clase a la que encapsula, el resto del software no necesita saber si está manejando el decorador o la clase original.

En el contexto de los árboles de comportamiento, un decorador es una tarea que tiene un único hijo, y que modifica su comportamiento. Como todos los nodos de los árboles de comportamiento comparten la misma interfaz, a nivel externo el resto del árbol no verá diferencia con respecto a si está lidiando con

el decorador o con la tarea original decorada.

Existe un amplio repertorio de tareas decoradoras.

El **inversor** es un decorador que invierte el resultado de la ejecución de la tarea que decora. Así, el *inversor* ejecuta su tarea decorada, y cuando ésta finaliza, invierte su estado. Si devuelve éxito, el *inversor* devolverá fracaso, y su devuelve fracaso, el *inversor* devolverá éxito.

El decorador **repetir** juega el papel de un bucle sin fin. Este decorador ejecuta su hijo, y cuando éste finaliza, vuelve a ejecutarlo. En realidad el nodo *repetir* no finaliza su ejecución nunca, de modo que nunca devuelve un estado de éxito o de fracaso.

El nodo **hasta fallo** se comporta de forma similar al nodo repetir, con la diferencia de que la ejecución del hijo se repite sólo mientras el hijo tenga éxito. En cuanto el hijo finaliza su ejecución en fracaso, el decorador *hasta fallo* finaliza, y devuelve éxito como estado de su ejecución (este decorador sólo puede devolver éxito como estado de ejecución).

El decorador **limitar** limita el número de veces que la tarea decorada se ejecuta. Este decorador es útil en situaciones donde se percibe que seguir repitiendo un comportamiento no tiene sentido. Por ejemplo, volviendo al árbol de la figura 2.6, el subárbol que intenta abrir una puerta embistiéndola podría ser controlado mediante un nodo *limitar*, para que el personaje intentara derribarla varias veces, intentos tras los cuales finalmente se daría por vencido.

2.4. Reutilización de Árboles

Una de las propiedades más interesantes de los árboles de comportamiento es su capacidad de reutilización. A diferencia de otras técnicas donde la reutilización es complicada, como las FSMs, la misma naturaleza de los árboles de comportamiento hace posible que sea fácil reutilizarlos en cualquier situación.

En un árbol de comportamiento, cualquier nodo representa un comportamiento. Salvo en las tareas de bajo nivel, estos comportamientos están a su vez compuestos de otros más sencillos (sus hijos en la jerarquía del árbol). En cualquier caso, es la raíz de cada subárbol lo que representa el comportamiento en cuestión. Así por ejemplo, el árbol de la figura 2.6 representa el comportamiento *Entrar en habitación*. Allí donde se quisiera que el personaje entrase en una habitación, bastaría reusar el árbol de la figura. Si dentro de un árbol más complejo una parte requiriera hacer que el personaje entrase en una habitación, bastaría colgar el árbol de la figura 2.6 desde su raíz en la parte correspondiente.

Para favorecer la reutilización de árboles de comportamiento, surge la necesidad de poder usar cualquier árbol sin necesidad de incrustarlo, literalmente, allí donde se necesite. Para ello, a cada árbol (comportamiento) de interés se le asigna un nombre único mediante el cual poder referenciarlo más tarde. En el caso del árbol de la figura 2.6, por ejemplo, se le podría asignar el nombre *EntrarEnHabitacion*. Es más, el nodo *selector* situado en el tercer nivel del árbol realiza un comportamiento bien definido: intentar abrir una puerta que está cerrada. Es por ello que, dado que es posible que fuera reutilizado en el futuro, se le podría asignar un nombre, tal como *AbrirPuerta*, a tal efecto.

Definida una *biblioteca* de árboles de comportamiento, es decir, un repositorio con todos los árboles de utilidad identificados por nombre, se puede hacer uso de una tarea conocida como **búsqueda de árbol**. La *búsqueda de árbol*

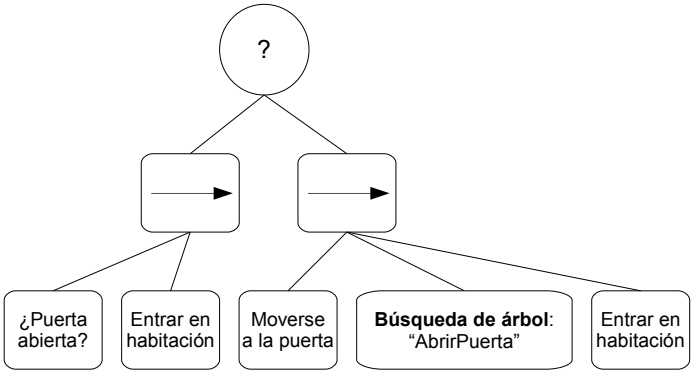

Figura 2.7: El árbol de comportamiento de la figura 2.6, simplificado

simplemente emula el comportamiento de un determinado árbol de comportamiento (identificado por nombre). Haciendo uso de este tipo de nodo, por ejemplo, el árbol de la figura 2.6 podría transformarse en el de la figura 2.7.

Capítulo 3

Combinando Darmok con Árboles de Comportamiento

A pesar de que Darmok muestra un rendimiento aceptable en general, presenta varios problemas que, en ciertos casos (y bastante frecuentes) pueden degradar su rendimiento de manera severa.

En el bajo nivel de la ejecución de sus planes, Darmok presenta un problema de reactividad. Darmok construye un plan global mediante la extracción de subplanes de su base de casos, alcanzando finalmente acciones de bajo nivel que son enviadas directamente al motor del videojuego para su ejecución inmediata. En escenarios de larga duración (macro-gestión), las acciones de bajo nivel se comportan lo suficientemente bien como para que el plan global evolucione exitosamente. Sin embargo, cuando las acciones deben ser alteradas o incluso canceladas en su totalidad debido a cambios rápidos en el estado del mundo, el hecho de que Darmok se adhiera a ellas deteriora su rendimiento: mientras la condiciones de vida de las acciones se mantengan activas, el plan seguirá ejecutándose, incluso si eventos recientes sugieren que la acción debería ser cancelada o modificada.

Esto se debe en parte al modo en que Darmok aprende sus planes a partir de las trazas que se le proporcionan. Si fuera capaz de aprender planes mejor estructurados, sería capaz de crear planes más complicados para las acciones de bajo nivel. Sin embargo, ello requeriría no sólo una abrumadora mayor cantidad de trazas, sino también un gran esfuerzo por parte del experto a la hora de anotar las trazas (en Darmok) o a la hora de definir el conocimiento dependiente del dominio necesario para el proceso de aprendizaje (Darmok 2). Desafortunadamente, la reactividad de dichos planes también estaría comprometida, dado que el modelo de ejecución de Darmok no se replantea los planes actualmente en ejecución a no ser que fallen.

Este problema tiene especial importancia cuando unidades individuales deben ser controladas a muy bajo nivel. Cuando a una unidad se le envía una orden (acción de bajo nivel) del plan que Darmok construye, puede darse el caso de que, mientras la ejecuta, debiera ser modificada de algún modo debido a cualquier tipo de evento relevante observado en el mundo. En juegos como StarCraft es fácil encontrar situaciones como ésta. Por ejemplo, en StarCraft, muchas unidades tienen habilidades especiales, como en el caso de los *altos templarios*. Los altos templarios tienen la capacidad de convocar *tormentas psiónicas* en ciertas posiciones del mapa. Una tormenta psiónica no es más que un hechizo que ocupa un pequeño área del mapa y que causa daños seve-

ros a las unidades situadas bajo ella. Las tormentas psiónicas pueden resultar tremendamente útiles durante el juego, ya que se pueden usar en ataque y defensa, destruyendo de forma masiva unidades enemigas. En ciertos casos pueden suponer la diferencia entre ganar y perder una batalla.

Cuando Darmok ordena a un alto templario convocar una tormenta psiónica en una posición $\{X, Y\}$ (es decir, genera la instrucción *ConvocarTormentaPsionica*), el alto templario irá a la posición de destino y lanzará la tormenta. Dado que Darmok ha llevado a cabo un paso previo de adaptación, es probable que $\{X, Y\}$ sea una posición similar a la de la traza original, la cual a su vez debería ser una región con muchas unidades enemigas y casi ninguna unidad aliada (para que así la tormenta no matara unidades aliadas). Sin embargo, mientras el alto templario se desplaza hasta $\{X, Y\}$, las unidades enemigas podrían haberse movido por los alrededores, con lo que lanzar la tormenta sería para entonces totalmente inútil, ya que ningún enemigo resultaría dañado. Es más, si muchas unidades aliadas se hubieran desplazado a $\{X, Y\}$, podrían ser destruidas por la tormenta. En estos casos sería conveniente que el alto templario pensara acerca del área en la que se le ha ordenado lanzar la tormenta.

Si bien es cierto que el problema podría solucionarse añadiendo nuevas condiciones de vida a la acción *ConvocarTormentaPsionica* (para que, por ejemplo, si la posición alcanzada no es apropiada para la tormenta la acción fuera cancelada), la idea detrás de Darmok es aprender mediante ejemplos, con la menor cantidad posible de conocimiento del dominio. Para que Darmok fuera muy reactivo, deberían definirse incontables condiciones de vida que modelasen todos los posibles escenarios, lo cual incrementaría de gran manera el esfuerzo requerido para la recopilación del conocimiento del dominio. Si, por ejemplo, el alto templario tuviera que huir de una eventual fuente de peligro (para que no lo matasen), se deberían definir nuevas condiciones de vida para la acción *ConvocarTormentaPsionica*. Además, aunque la adición de más condiciones de vida puede dar lugar a una mayor reactividad por parte de Darmok, el hecho de que se cancelen las acciones supone llevar a cabo una replanificación, la cual podría no ser necesaria.

En general, en dominios donde se requiere una capacidad de reacción rápida a bajo nivel, Darmok ve empeorado su rendimiento [24], como es el caso de videojuegos como *S2* (parecido a StarCraft) o *BattleCity* (un juego de acción más que de estrategia). Darmok fue diseñado inicialmente para competir en juegos de estrategia, donde el requerimento de la capacidad de reactividad puede ser elevado en ciertos dominios, como el de StarCraft o S2. Es por ello que su baja capacidad reactiva impide que en ciertos escenarios rinda a nivel humano, llegando incluso a no superar a las IAs predefinidas.

Paradójicamente, a alto nivel Darmok encuentra situaciones en las que no rinde adecuadamente debido justamente a un comportamiento excesivamente reactivo. Cuando Darmok aprende planes a partir de las trazas que se le proporcionan, puede darse el caso de que algunos de los *planes objetivo* aprendidos no tengan una estructura adecuada. Este problema está en parte relacionado con el modelo de ejecución de Darmok. Darmok asocia planes a objetivos. Cuando alguna de las acciones individuales o de los planes objetivo que componen el plan principal falla, el plan principal también se marca como fallido y Darmok lo descarta, marcando su objetivo abierto nuevamente. Como resultado Darmok intentará encontrar un plan diferente para el objetivo, que pasará a

ejecutar.

En ciertos escenarios este comportamiento no conlleva un buen resultado. Por ejemplo, si un plan objetivo *DestruirAlEnemigo* se usa para acabar con todas las unidades del enemigo, los planes para dicho objetivo estarán compuestos de muchas acciones o subplanes, todos ellos con el propósito de destruirlas. En el fragor de la batalla, sin embargo, en realidad se espera que las unidades fallen las acciones que intentan llevar a cabo, ya que en una batalla hay multitud de ocasiones que pueden impedirles completar sus tareas (por ejemplo, podrían ser destruidas). Así, en estos casos, Darmok seguirá fallando los planes que obtenga de la base de casos para el objetivo *DestruirAlEnemigo*, debido al fallo continuado de las acciones o subplanes que lo componen. Al final, después de intentar varios planes, Darmok acaba haciendo algo que tiene algún sentido, pero la calidad de la solución, por supuesto, no es en absoluto buena. Ésta es de hecho la principal razón por la cual Darmok no es bueno en combate, necesitando generalmente tener el doble de unidades que el enemigo para poder derrotarlo [25].

Por último, hay que resaltar el hecho de que la planificación basada en casos para juegos de estrategia requiere un gran esfuerzo por parte del experto si se quiere refinar escenarios muy específicos. Si el experto detecta una carencia en la base de casos, debería proporcionar un nuevo plan que aprender. Desafortunadamente, el único modo de hacerlo consiste en jugar una partida que simule el escenario concreto a aprender, lo cual puede llegar a ser extremadamente difícil debido a la complejidad y aleatoriedad inherente a los juegos de estrategia.

Nuestra propuesta consiste en hacer uso de árboles de comportamiento para superar los problemas presentados. En primer lugar, dado que los árboles de comportamiento permiten al diseñador definir comportamientos a bajo nivel, pueden usarse para modelar escenarios muy complejos (además, los diseñadores están acostumbrados a diseñar comportamientos complejos). En segundo lugar, pueden emplearse para definir acciones de bajo nivel, es decir, cómo el sistema debe comportarse ante las acciones enviadas por Darmok. Por último, pueden usarse para modelar planes para objetivos en situaciones en las que es conveniente tener un mejor control que el que Darmok tiene por defecto. La idea es extender la arquitectura de Darmok mediante una capa táctica basada en árboles de comportamiento, y que se encargue de gestionar algunas acciones de bajo nivel así como algunos planes de tipo objetivo. Cuando Darmok genera una acción de bajo nivel o un plan objetivo, se lleva a cabo una decisión acerca de si es conveniente o no que sea gestionada mediante un árbol de comportamiento. La idea es proporcionar árboles que, en algunos escenarios, se espera que se comporten de manera correcta como substitutos de acciones de bajo nivel o planes objetivo.

3.1. Capa Táctica para la Gestión de Acciones de Bajo Nivel

Las acciones de bajo nivel, es decir, aquellas generadas por Darmok en cada ciclo del juego, deben ser ejecutadas directamente en la API del juego. Con el fin de tener un mejor control sobre ellas, proponemos el uso de árboles de comportamiento para implementarlas. Inicialmente podría pensarse que, para

cada tipo de acción, tales como *AtacarEnemigo* o *Moverse*, podría haber un árbol de comportamiento que las implementara. Serían en ese caso árboles de comportamiento orientados por acción, es decir, árboles de comportamiento que persiguen un determinado objetivo (la acción en sí), pero también con la capacidad de cambiar su comportamiento general en caso de ser necesario. Esto es justamente lo que explicábamos anteriormente: cuando a una unidad se le da algún tipo de orden, debería intentar cumplirla, pero también debería ser capaz de cambiar su comportamiento en ciertos casos. Sin embargo, no es un enfoque realista esperar que un único árbol de comportamiento proporcione buenos resultados a la hora de controlar cada tipo de acción (por ejemplo, las acciones *AtacarEnemigo* o *Moverse* antes mencionadas) en todas las situaciones posibles. Nuestro enfoque consiste en disponer de varios árboles de comportamiento para cada tipo de acción. La capa táctica se encarga de asociar un árbol de comportamiento concreto, de la lista de posibles árboles, a cada acción. Para realizar esta decisión, a nivel de diseño, el diseñador debe especificar una descripción del estado del mundo en aquellas situaciones en las que se espera que el árbol muestre un buen comportamiento. En tiempo de ejecución, la capa táctica comprueba la similitud entre el estado actual del mundo y los estados proporcionados por todos los árboles de comportamiento para el tipo de acción, y el árbol con el estado más cercano es seleccionado. En el caso de que ningún árbol de comportamiento pueda ser recuperado (por ejemplo, si el árbol más cercano no supera un determinado umbral de distancia), la acción se envía directamente a la API del juego (en cuyo caso ningún árbol de comportamiento le sería asociado).

La figura 3.1 muestra la arquitectura de la capa táctica a bajo nivel. Para cada acción que Darmok produce, la capa táctica la procesa. La capa táctica pide a una *Base de BTs* externa un árbol de comportamiento para gestionar la acción. Dependiendo del tipo de acción, la Base de BTs recupera un conjunto de árboles de comportamiento candidatos (los diseñados para ese tipo de acción); entonces, compara el estado del mundo actual con los estados de los árboles de comportamiento en el conjunto de candidatos, y aquél cuyo estado es más cercano es devuelto. Este árbol de comportamiento es entonces incorporado a una *Piscina de BTs*, compuesta por todos los árboles de comportamiento que la capa táctica está gestionando actualmente. Tanto la capa táctica como los árboles de comportamiento conocen en todo momento cuál es el estado del juego, para así tener un mayor grado de control de lo que en él acontece. En el caso de que un árbol de comportamiento no pudiera ser recuperado de la Base de BTs, la acción de bajo nivel sería enviada finalmente a la API del juego. Es importante notar que esta parte de la capa táctica es completamente independiente de Darmok, es decir, no requiere modificación de la arquitectura de Darmok.

3.2. Capa Táctica para la Gestión de Planes Objetivo

En aquellos casos en los que Darmok no ha sido capaz de aprender planes efectivos para algunos objetivos, los árboles de comportamiento se pueden usar como alternativa. La idea es similar a la de la gestión de las acciones de bajo nivel. Un experto construye árboles de comportamiento. Estos árboles

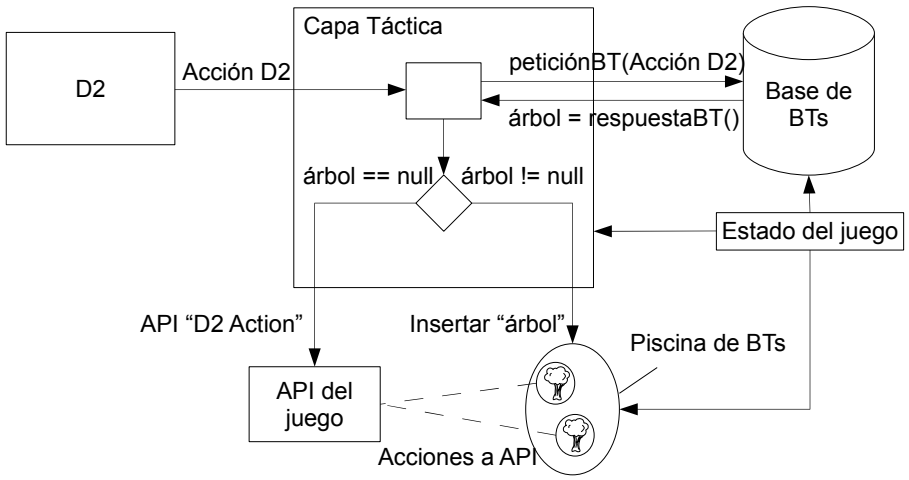

Figura 3.1: Capa táctica de bajo nivel

contienen una descripción del estado del juego que especifica cuándo es adecuado hacer uso de ellos. Cuando un árbol de comportamiento es recuperado con un objetivo particular, serán el árbol en sí, junto con la capa táctica, los encargados de gestionar el objetivo original.

La figura 3.2 muestra la arquitectura de la capa táctica a alto nivel. Inicialmente, Darmok procede como de costumbre. Sin embargo, cuando se detecta un objetivo dentro de un plan, el módulo de expansión de Darmok comprueba si el objetivo debe ser gestionado por un árbol de comportamiento. Para realizar esta comprobación, Darmok realiza una petición a la Base de BTs, del mismo modo que en el escenario de las acciones de bajo nivel. Si la Base de BTs puede recuperar un árbol de comportamiento para el objetivo y estado del juego actual, el plan objetivo actual (Obj. 3 en la figura) es reemplazado por un *Plan de BT*, y el árbol de comportamiento es marcado para ser enviado fuera de Darmok del mismo modo que las acciones de bajo nivel que genera en cada ciclo del juego. Un Plan de BT no es más que un plan de Darmok cuya ejecución no es gestionada por Darmok, sino por un árbol de comportamiento externo. En lo que a Darmok respecta, no importa si el plan es gestionado fuera suya, ya que, mientras pueda proporcionar la misma interfaz que la de un plan estándar de Darmok, éste sabrá cómo gestionarlo. De este modo, en cada ciclo del juego, Darmok no sólo genera acciones de bajo nivel (a ser procesadas como se ha descrito en la sección 3.1), sino que también genera árboles de comportamiento que deben ser ejecutados por la capa táctica (y que son insertados en la Piscina de BTs). En caso de que un árbol de comportamiento no pueda ser obtenido de la Base de BTs, Darmok procede como siempre, es decir, expandiendo el objetivo mediante un plan obtenido de la base de planes. La figura 3.2 se corresponde con la situación en la que un árbol de comportamiento (*árbol* en la figura) es recuperado para la ejecución del Obj. 3.

3.3. Arquitectura Global

En las secciones 3.1 y 3.2 se ha descrito cómo se puede hacer uso de árboles de comportamiento para gestionar tanto acciones de bajo nivel generadas por Darmok así como planes objetivo. La parte central de la arquitectura es la capa

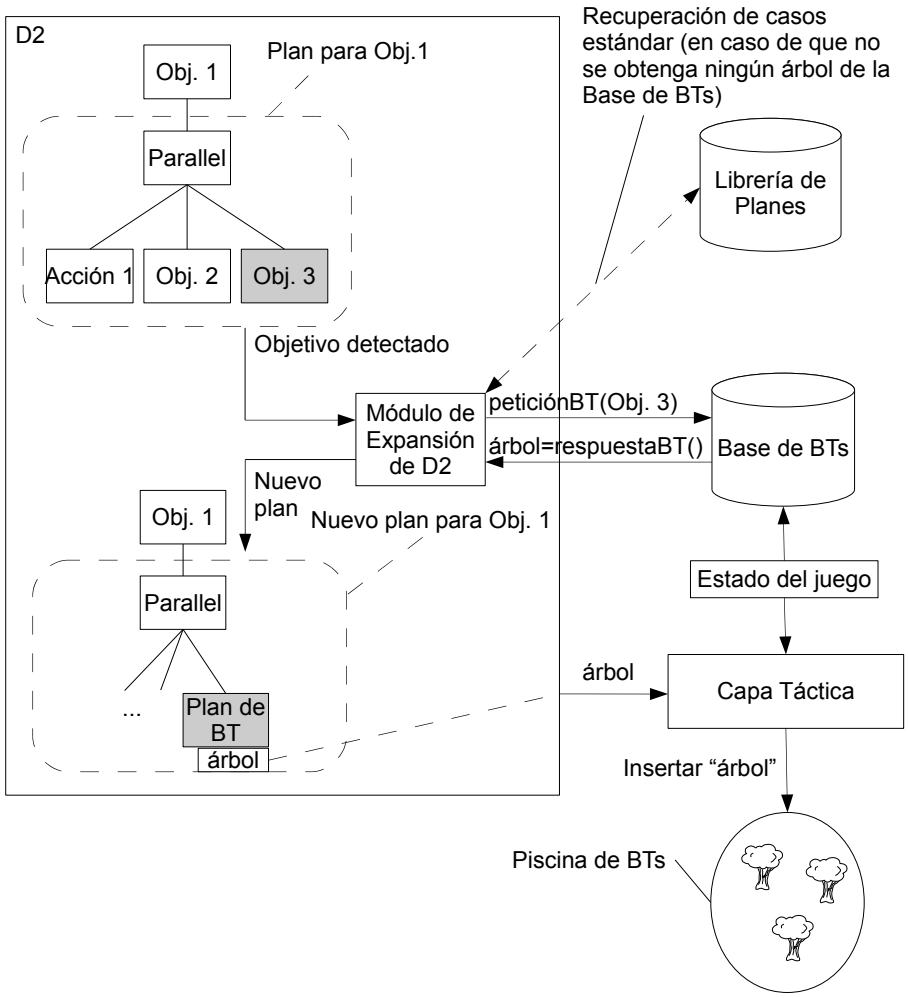

Figura 3.2: Capa táctica de alto nivel

táctica mencionada con anterioridad. La capa táctica gestiona una piscina de árboles de comportamiento (la Piscina de BTs). Esta piscina contiene todos los árboles de comportamiento actualmente en ejecución. Cuando Darmok genera una nueva acción de bajo nivel, la capa táctica pide a la Base de BTs un árbol de comportamiento para la acción dado el estado actual del juego. Si puede encontrar una, la capa táctica la inserta en la Piscina de BTs. Si no, la acción es enviada directamente a la API del juego. Además, si el plan que Darmok está ejecutando contiene planes objetivo adecuados para ser gestionados por árboles de comportamiento (como el Obj. 3 de la figura 3.2), se construyen árboles de comportamiento para gestionarlos, los cuales son consecuentemente generados por Darmok junto con las acciones de bajo nivel. Estos árboles de comportamiento son insertados entonces en la Piscina de BTs.

En cada ciclo del juego, la capa táctica da a los árboles de comportamiento de la Piscina de BTs algo de tiempo para que se ejecuten. Es en ese momento cuando los árboles envían acciones a la API del juego. En lo que respecta a los árboles de comportamiento de acciones de bajo nivel, éstos son finalizados por la capa táctica cuando las condiciones de fallo o de éxito (conforme a la semántica de Darmok) de las acciones que representan se hacen ciertas, dado que es en ese momento cuando Darmok espera que dejen de ejecutarse para que pueda continuar con la evolución del plan global. Con respecto a los árboles de comportamiento de planes objetivo, éstos interaccionan con Darmok a través de Planes de BT especiales. Mientras estos planes proporcionen un modo de comprobar sus condiciones de éxito y sus condiciones de fracaso, Darmok puede interaccionar con ellos de manera normal. Las condiciones de éxito de estos planes son las mismas que la del objetivo que representan. El aspecto relevante son las condiciones de fallo. En una situación normal, el plan sería marcado como fallado tan pronto como una acción o subplan de él fallase. En este caso, sin embargo, dado que toda la ejecución del plan es gestionada por un árbol de comportamiento, las condiciones de fallo son establecidas por el mismo árbol, permitiendo así finalizar el plan cuando se considere oportuno.

3.4. Recuperación de Árboles mediante Métrica de Similitud

Un aspecto importante de la arquitectura propuesta es el de asociar, a cada árbol, un estado del mundo. Así, cuando la capa táctica lleva a cabo la recuperación de un árbol, elige sólo aquellos cuyo estado del mundo es similar al estado actual. En [28] y en [8] se propone un mecanismo de recuperación de comportamientos basado en CBR que implementa esta idea (aplicándolo en un caso a máquinas de estado finitas jerárquicas, y en otro a árboles de comportamiento). La recuperación de árboles de comportamiento en nuestra arquitectura se puede llevar a cabo siguiendo un esquema similar, pero simplificado respecto al propuesto por los autores.

A cada *caso* (árbol de comportamiento), se le asocia un conjunto de elementos *descriptores* que describen el estado del mundo en que es apropiado usar el árbol. Un descriptor representa una característica del mundo, con un valor (numérico o simbólico) asociado. Además, se le asocia una *clase* que indica el tipo de comportamiento representado (las clases pueden organizarse en taxonomías).

Cuando se quiere recuperar un árbol, se construye una *consulta* que indica el tipo de árbol que se desea. Dicha consulta incluye tanto la clase del comportamiento que se desea obtener como un subconjunto de descriptores del estado del mundo que se consideren relevantes a la hora de realizar la consulta (es decir, qué características del estado del mundo se deben utilizar a la hora de obtener el árbol).

Dada una consulta Q y un caso C, se computa la similitud entre Q y C según $sim(Q, C)$:

$$sim(Q,C) = \begin{cases} 0 & Q.clase \neq C.clase \\ sim_{atr}(Q,C) & Q.clase = C.clase \end{cases}$$

$$sim_{atr}(Q,C) = \sum_{d \in D(Q,C)} w_d \cdot sim_{loc}(Q_d, C_d)$$

$$D(Q,C) = Q.descriptores \cap C.descriptores$$

$$sim_{loc}(Q_d, C_d) = 1 - \frac{|Q_d.valor - C_d.valor|}{tam_d}$$

Los pesos w_d asociados a cada descriptor miden la importancia que se le da a cada uno a la hora de calcular la medida de similitud, tienen un valor en el intervalo $[0, 1]$, y la suma de todos ellos debe ser 1 ($\sum_{d \in D} w_d = 1$). tam_d representa el tamaño del intervalo para los valores del descriptor d (valores válidos dentro del sistema). Así se fuerza a que la medida de similitud calculada esté normalizada en el intervalo $[0, 1]$. Es importante recordar que los valores de los descriptores pueden ser tanto numéricos como simbólicos, en cuyo caso se les debe asignar un valor numérico a la hora de realizar el cómputo de distancias.

Dada una consulta Q, la medida de similitud respecto a un caso C permite recuperar aquel caso más similar a la consulta dada, usando para ello el estado del mundo actual y el estado del mundo al cual el caso recuperado debería parecerse (especificado en los descriptores de la consulta).

En el contexto de la arquitectura de la capa táctica planteada, cuando se lleva a cabo una recuperación de árboles de comportamiento, se debe construir una consulta adecuada para que se recupere el árbol más apropiado para ello. Así, a la hora de recuperar un árbol B que gestione una acción de bajo nivel A (capa táctica para las acciones de bajo nivel), se construye una consulta cuya clase es la misma que la de A, y cuyos descriptores recogen parte del estado global del juego así como del estado local de la unidad que va a ejecutar la acción. El estado local de la unidad es muy importante, ya que en general la estrategia a seguir depende del entorno inmediato de la entidad que ejecuta la acción. No es así el caso de los árboles recuperados por la capa táctica de alto nivel (planes objetivo), ya que a la hora de gestionar un objetivo más general es probable que el estado global del juego influya en mayor medida (lo cual no descarta que también se usen descriptores más específicos).

3.5. Escenario

En esta sección planteamos dos escenarios en los que nuestra arquitectura mejora el rendimiento de Darmok. Uno de ellos es un escenario de acciones de bajo nivel, mientras que el otro es un escenario de planes objetivo.

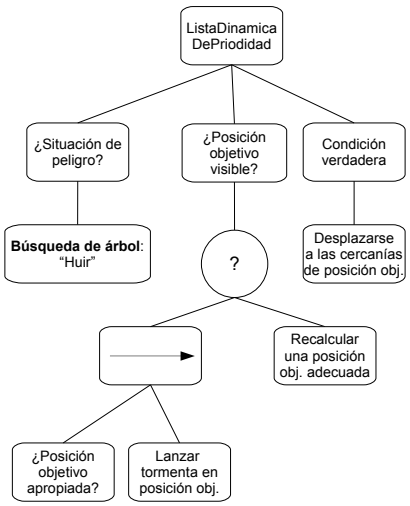

Figura 3.3: Árbol de comportamiento para la acción *ConvocarTormentaPsionica*

3.5.1. Escenario de Acciones de Bajo Nivel

En el escenario de acciones de bajo nivel, fijémonos en la acción *ConvocarTormentaPsionica* definida con anterioridad.

La figura 3.3 muestra un árbol de comportamiento que implementa el modelo reactivo explicado en la sección 3.1: el alto templario no sólo tiene constancia de la posición objetivo sobre la que tiene que convocar la tormenta, sino que también reacciona ante situaciones de peligro. Este árbol de comportamiento también define un estado del juego que especifica cuándo es conveniente hacer uso de él. Dado que éste es un árbol muy estándar, podría ser usado en muchos escenarios, si bien probablemente estaría mejor adecuado para escenarios en los que tanto las fuerzas aliadas como enemigas estuvieran balanceadas. En ese caso, el alto templario debería intentar no dañar unidades aliadas así como huir en caso de peligro, para no perder ventaja.

En otros escenarios, por ejemplo cuando el ejército aliado supera en gran medida al enemigo, los altos templarios podrían comportarse de manera más agresiva, o incluso temeraria, así que se podrían implementar nuevos comportamientos (árboles de comportamiento) con distintos estados del juego asociados.

Cuando Darmok genera la orden *ConvocarTormentaPsionica*, la capa táctica la procesa. Si en el estado actual del juego tanto las fuerzas aliadas como enemigas están balanceadas, el árbol de la figura 3.3 podría ser recuperado para gestionar dicha acción. En ese caso, sería insertado en la Piscina de BTs para su uso posterior. Si no se encontrase ningún árbol de comportamiento, la acción *ConvocarTormentaPsionica* sería enviada de forma directa a la API del juego para ser ejecutada.

3.5.2. Escenario de Plan Objetivo

Cuando Darmok gestiona algunos de sus objetivos, no es de esperar que se comporte correctamente. Tómese por ejemplo el caso del objetivo *DestruirAlEnemigo* mencionado anteriormente. Cuando Darmok construye planes para dicho objetivo, éstos tomarán la forma de numerosas acciones de bajo nivel, tal vez organizadas en estructuras de tipo paralelo o secuencia. Aunque los planes pudieran parecer muy complejos, todos compartirían la misma estructura,

estando compuestos de numerosas acciones de bajo nivel sin apenas ningún subobjetivo. Cuando una acción o subplan falle, el plan entero fallará, y un nuevo plan deberá ser recuperado para la gestión del objetivo *DestruirAlEnemigo*. Dado que este problema se repetirá continuamente, Darmok no se comportará de manera adecuada en esta situación.

Para solventar el problema, podemos definir árboles de comportamiento para gestionar el objetivo *DestruirAlEnemigo* en diversos escenarios. Por ejemplo, se podría construir un árbol a usar en situaciones donde el ejército aliado superase con creces al enemigo. En tales situaciones, un buen modo de actuar consiste en ordenar que todas las unidades aliadas ataquen de manera simultanea los edificios y unidades enemigas hasta que sean destruidos por completo. Al igual que en el escenario de bajo nivel, el árbol de comportamiento contendría una descripción del estado del mundo en el que debería ser usado. Cuando el objetivo *DestruirAlEnemigo* es generado en el plan que Darmok construye, el módulo de expansión de Darmok intentará encontrar un árbol de comportamiento apropiado en la Base de BTs, para ello usando el estado actual del mundo así como el estado asociado al árbol. Si las unidades aliadas superaran con creces a las enemigas, el árbol de comportamiento descrito podría ser recuperado y enviado fuera de Darmok, junto con las acciones que normalmente produce. A partir de entonces, sería manejado por la capa táctica.

Capítulo 4

Java Behaviour Trees, un *Framework* para el Desarrollo de Árboles de Comportamiento en Java

Durante el diseño del sistema basado en árboles de comportamiento para la extensión de Darmok, nos encontramos con el problema de hacer uso de algún framework de árboles de comportamiento que implementara, al menos, un modelo suficientemente potente como para que se pudieran diseñar comportamientos elaborados. Además, dicho modelo debía estar implementado preferiblemente en Java, ya que al ser éste el lenguaje en el que Darmok está desarrollado, se conseguiría una mejor integración.

Hasta donde llega nuestro conocimiento, no existe actualmente ningún framework de árboles de comportamiento basado en licencias de software libre (tales como *GNU GPL* o similares) e implementado en Java. Es por ello que decidimos implementar por nuestra cuenta nuestro propio framework de árboles de comportamiento en Java. El resultado es *Java Behaviour Trees* (JBT), un framework que permite al desarrollador definir de forma rápida y sencilla árboles de comportamiento, almacenarlos en ficheros XML estándar (para una posterior reutilización), y ejecutarlos como código nativo Java. JBT está liberado bajo licencia GNU GPLv3, pudiendo accederse a él a través de la página oficial de *Sourceforge*, http://sourceforge.net/projects/jbt/. Para una descripción detallada de cómo funciona JBT, referimos al lector al apéndice A.

4.1. Características Principales de JBT

A la hora de implementar JBT se tuvieron que tomar ciertas decisiones de diseño que determinarían cómo todo el framework sería implementado. Además, se tuvieron en cuenta problemas comunes asociados al uso de árboles de comportamiento, para los cuales se intentó dar una solución adecuada.

4.1.1. Modelo Conducido por Ticks

JBT implementa un modelo de árboles de comportamiento *conducido por ticks*. Esto significa que un árbol de comportamiento es evaluado sólo a través de ticks, donde un *tick* representa una cantidad de tiempo de CPU que se le

proporciona al árbol para que se ejecute. Así, en cada ciclo de la IA del videojuego, una entidad externa *hace tick* en el árbol, para que éste pueda actualizar su estado. En concreto, cada tick hace que los nodos del árbol evalúen si han finalizado o no, para que consecuentemente el árbol evolucione. En el modelo abstracto conceptual planteado en la sección 2.1, un tick se corresponde con una llamada al método *update()* de la clase *BTNode*.

El enfoque más simple para implementar el modelo conducido por ticks es justamente el de la sección 2.1: el nodo raíz recibe el tick (llamada al método *update()*), el cual es propagado de manera recursiva hacia sus descendientes de acuerdo con la semántica de cada nodo. Sin embargo, éste es un proceso ineficiente, sobretodo para árboles de comportamiento suficientemente grandes, ya que en general la mayor parte de los nodos del árbol están esperando a que sus hijos finalicen su ejecución. Por tanto, el que dichos nodos reciban ticks se convierte en un desperdicio de tiempo CPU, ya que, al menos que sus hijos hayan finalizado su ejecución, no harán nada al recibir el tick. En realidad, en general, sólo un conjunto muy reducido de nodos debería recibir ticks en cada ciclo de IA. Como resultado, JBT implementa un modelo en el que existe una lista de nodos *tickeables*. Sólo los nodos en esta lista pueden recibir ticks.

4.1.2. Modelo Independiente de Ejecución

Cuando se ejecutan árboles de comportamiento surge la necesidad de reutilizar un mismo árbol en varias entidades del juego, es decir, hacer que varias entidades del juego sigan el comportamiento definido en un mismo árbol.

El enfoque más simple a través del que resolver esta situación consiste en partir de un modelo de árbol *arquetipo*, y hacer una copia del árbol para cada entidad del juego que necesite usarlo. El problema de este método es que, si son numerosas las unidades que hacen uso del árbol, para cada una de ellas existiría una copia íntegra de éste. En caso de que el árbol sea grande y/o muchas entidades del juego lo requieran, el consumo en memoria podría dispararse.

Una solución más adecuada pasa por definir el árbol a nivel conceptual, de modo que un *intérprete* de árboles se encarga de leerlo e ir ejecutándolo conforme se necesite. Según este enfoque, que es el adoptado por JBT, existe un intérprete por cada entidad del juego que necesita ejecutar el árbol. Cada intérprete recibe una referencia al modelo del árbol que se quiere ejecutar, de modo que todos comparten el único modelo definido para el árbol. Así, aunque para cada entidad exista un intérprete, a nivel global existe una única copia del árbol, con el consiguiente ahorro en memoria.

Se hace así una clara distinción entre el árbol que está siendo ejecutado, el *modelo*, y cómo está siendo ejecutado, su *ejecución*. Para cada comportamiento, en JBT se distingue entre el *Árbol Modelo* que lo define y el cómo es ejecutado. El *cómo* es gestionado por el intérprete de árboles, el *Ejecutor de Árbol*. El Ejecutor de Árbol analiza un Árbol Modelo y lo simula a través de los ticks que recibe de una entidad externa al framework. Por cada tick, el Ejecutor de Árbol avanza en la simulación del árbol.

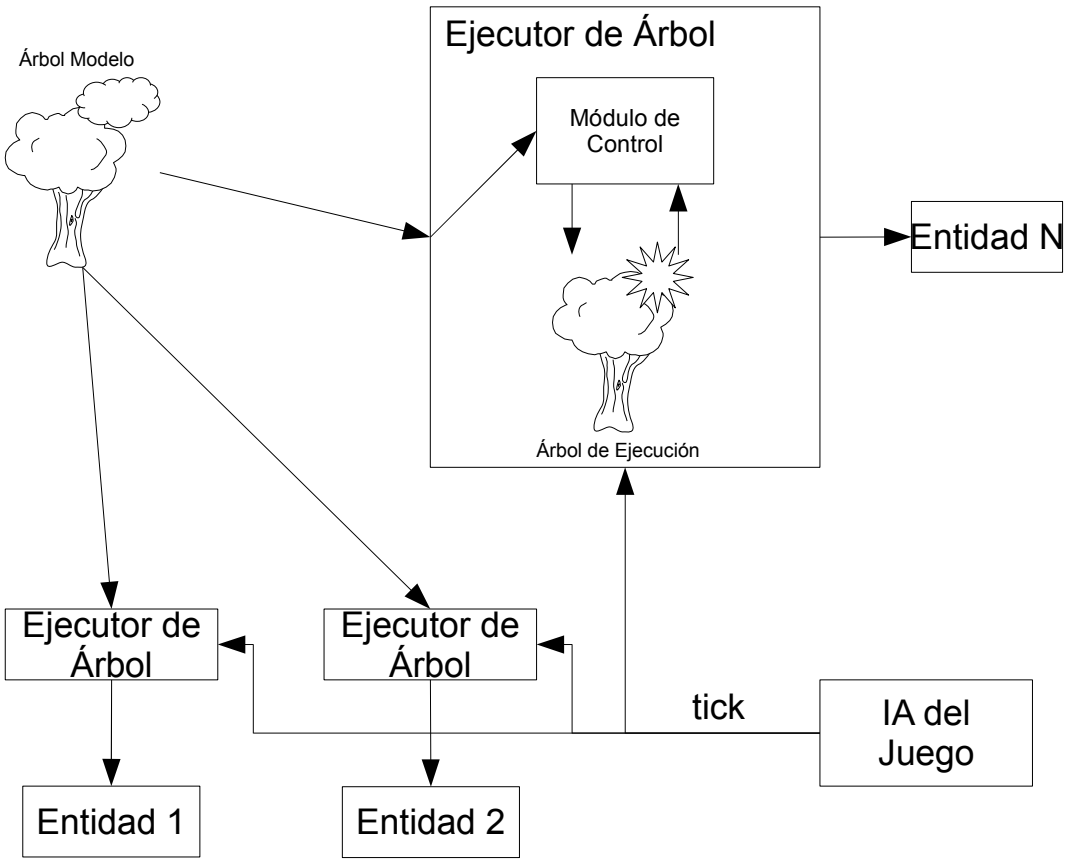

Figura 4.1: Arquitectura global de JBT

4.1.3. Arquitectura Global

La figura 4.1 muestra la arquitectura general de JBT. Cada comportamiento está definido a través de un Árbol Modelo. Para cada entidad (Entidad 1, Entidad 2, ..., Entidad N) que quiere ejecutar el comportamiento existe un Ejecutor de Árbol. Cada Ejecutor de Árbol hace uso del Árbol Modelo, lo interpreta y lo ejecuta, simulando así el comportamiento definido en él.

Internamente, el Ejecutor de Árbol construye un *Árbol de Ejecución* parcial que contiene sólo aquellos nodos envueltos en la ejecución del árbol. Sin embargo, esta mecánica interna es ocultada al usuario del Ejecutor de Árbol, que simplemente percibe que el Árbol Modelo avanza en su ejecución a cada tick recibido por el Ejecutor de Árbol.

Por último, es el módulo de IA del videojuego el encargado de hacer tick a cada Ejecutor de Árbol, para que éstos avancen en la ejecución del árbol de comportamiento.

4.1.4. Modelo de Árboles de Comportamiento

JBT implementa un modelo de árboles de comportamiento basado principalmente en el recogido en [19], pero extendido con el concepto de guardas [8].

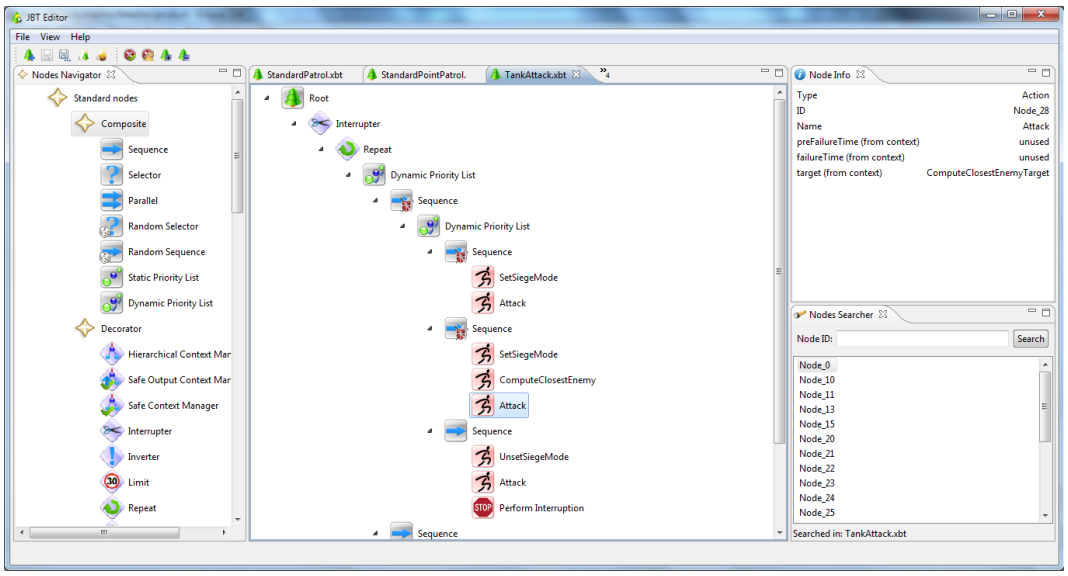

Figura 4.2: JBT Editor

4.2. JBT Editor

Para facilitar la creación de árboles de comportamiento, JBT incluye un editor gráfico de árboles de comportamiento, *JBT Editor* (figura 4.2).

El JBT Editor permite al usuario crear árboles de comportamiento de manera rápida y sencilla. Mediante un sencillo mecanismo de *arrastrar y soltar*, el usuario puede añadir cualquier tipo de nodo a sus árboles de comportamiento. JBT Editor permite además cargar ficheros XML que definen las acciones y condiciones de bajo nivel dependientes del dominio, las cuales pueden ser usadas como cualquier otro tipo de nodos a la hora de crear árboles de comportamiento. Los árboles creados pueden ser exportados a ficheros XML que son posteriormente analizados por JBT para construir clases Java que representan a los árboles en cuestión.

Capítulo 5

Estudio Experimental

Inicialmente, este trabajo se planteó en el contexto del concurso de StarCraft organizado por *Expressive Intelligence Studio* en la Universidad de California, Santa Cruz, en el 2010, motivo por el cual se decidió implantar para su uso en el videojuego StarCraft.

En este capítulo se recoge el estudio experimental llevado a cabo para evaluar la arquitectura híbrida entre Darmok y árboles de comportamiento propuesta. Se describe el dominio elegido (StarCraft), la tecnología empleada para la comunicación con éste desde un programa Java, los experimentos planteados y los resultados obtenidos.

5.1. Dominio, StarCraft

El dominio elegido para testear la arquitectura propuesta es StarCraft [5]. StarCraft (ver figura 5.1) es un juego de estrategia en tiempo real que ha cosechado millones de jugadores alrededor del mundo en la última década, y que se ha convertido, posiblemente, en el juego de estrategia más famoso de la historia.

Como en la mayoría de los juegos de estrategia en tiempo real, en StarCraft el jugador controla un ejército (a elegir entre tres razas distintas), siendo su deber el de derrotar a uno o varios oponentes.

Inicialmente, el jugador dispone de un edificio central y cuatro *trabajadores*. Los trabajadores son usados para extraer recursos del mapa (bien *minerales*, bien *gas vespeno*), los cuales se emplean para construir edificios y unidades. Los trabajadores también se usan para la construcción de edificios. Para vencer al enemigo (o enemigos), el jugador debe disponer de un ejército lo suficientemente bien armado (generalmente compuesto de tropas de diversos tipos), y para construir el ejército requiere de edificios con los que entrenar a las tropas adecuadas. Además, ciertos edificios pueden emplearse para mejorar la tecnología del armamento de las unidades, lo cual puede otorgar una gran ventaja durante el desarrollo de la partida.

Al igual que en la mayoría de los videojuegos actuales, StarCraft emplea una IA codificada manualmente, predecible y que puede ser derrotada fácilmente una vez se aprende su patrón de juego. En concreto, StarCraft dispone de una serie de estrategias predefinidas, de modo que al inicio de cada partida la IA de la máquina elige una para el resto de la partida.

StarCraft dispone de tres razas a elegir a la hora de jugar una partida: *Terran*, que juegan el papel de los humanos, *Protoss*, que juegan el papel de

Figura 5.1: Una partida de StarCraft, donde se enfrentan dos ejércitos

civilización alienígena avanzada tecnológicamente, y *Zerg*, que juegan el papel de alienígenas no inteligentes que hace uso de sus fuerza física para vencer al enemigo (la figura 5.1 muestra una batalla entre Zerg y Terran). Cada raza tiene distintos tipos de unidades y diferentes habilidades especiales, con lo que el tipo de estrategia que se puede seguir con una generalmente no es aplicable al resto.

5.2. Comunicación con StarCraft

Para poder testear la arquitectura propuesta se necesita de un método de comunicación con StarCraft, para poder enviar desde Darmok las instrucciones que se quieran ejecutar dentro del juego.

Chaoslauncher [14] es una aplicación gráfica que permite al usuario ejecutar StarCraft con varios *plugins* adicionales, así como inyectar en el juego bibliotecas DLL.

Para poder comunicar una IA con el juego se hace uso de BWAPI [35], una API escrita íntegramente en C++ que permite compilar bibliotecas DLL que, inyectadas a través de Chaouslauncher, son capaces de acceder al estado del juego y modificarlo.

Con BWAPI el usuario puede consultar el estado completo del juego en StarCraft. Por ejemplo, puede consultar cuántas unidades hay actualmente en el mapa, qué órdenes están ejecutando, y cuál es su estado de salud. BWAPI no sólo accede a StarCraft en modo lectura, sino que permite modificar su estado mediante el envío de órdenes a las unidades del mapa. Cualquier tipo de orden que un jugador podría enviar a las unidades durante una partida, pueden también enviarse a través de BWAPI.

La figura 5.2 muestra el esquema de integración de BWAPI con StarCraft. El usuario construye un módulo de IA haciendo uso de la biblioteca BWAPI. Es en dicho módulo donde el usuario define cómo va a funcionar su IA (accediendo

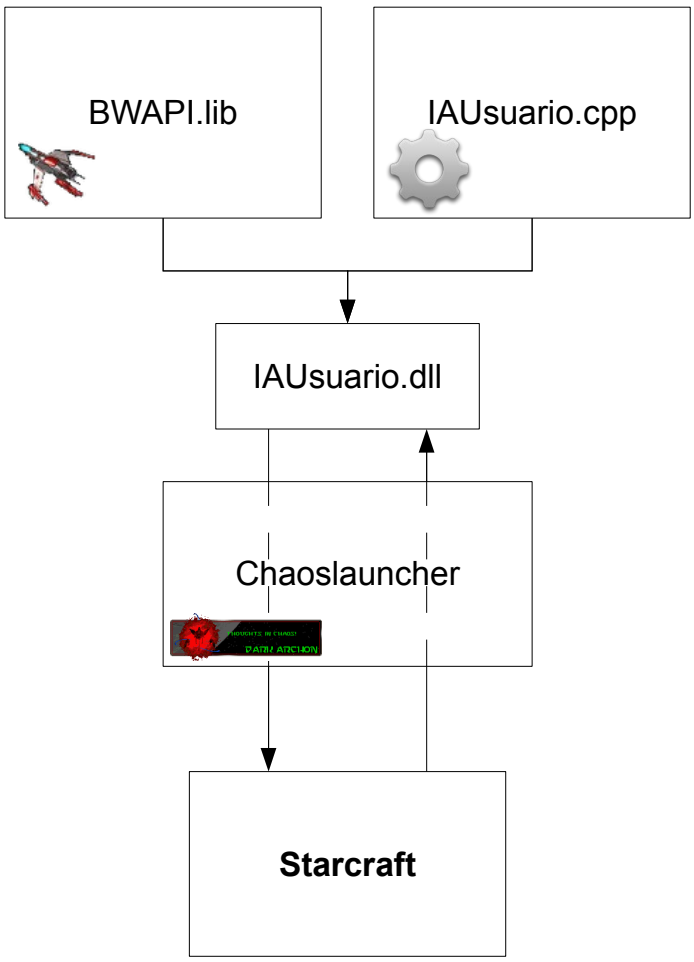

Figura 5.2: Integración de BWAPI con StarCraft

al estado del mundo y enviando órdenes al juego). Luego compila dicho módulo en una biblioteca DLL, la cual es finalmente inyectada por Chaoslauncher al proceso de StarCraft. Cuando el usuario comience una partida de StarCraft, la DLL se pondrá en funcionamiento y su módulo de IA comenzará a ejecutarse.

5.2.1. Comunicación en Java

Si bien BWAPI proporciona una interacción completa con la API de StarCraft, su principal inconveniente es la dependencia del lenguaje de programación C++.

Darmok, así como JBT, están implementados en Java, lo cual hace imposible su integración directa con BWAPI.

Además, implementar la IA en C++ lleva consigo los problemas asociados a este lenguaje. El principal es que la DLL inyectada por Chaoslauncher interfiera con el espacio de direcciones de StarCraft, rompiendo el proceso, y teniendo que reiniciar StarCraft, lo cual podría ralentizar en gran medida la experimentación.

Para resolver estos problemas se plantea el uso de un proceso externo que se comunique mediante *sockets* con StarCraft. Así, la DLL que Chaoslauncher inyecta en el proceso de StarCraft se encarga de leer constantemente el estado del mundo y enviarlo fuera, a través de un socket. Por otro lado, un proceso externo, que puede estar implementado en cualquier lenguaje, se encarga de leer del socket el estado del mundo. Este proceso externo, además, envía al

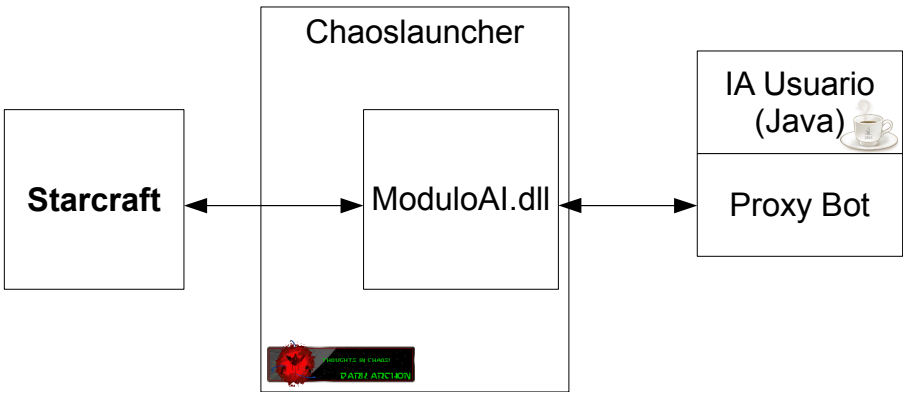

Figura 5.3: Proxy Bot para la comunicación Java con Starcraft

socket las acciones que se quieren ejecutar en StarCraft, las cuales son leídas y ejecutadas por la DLL inyectada a través de Chaoslauncher.

Para la experimentación se hizo uso de una versión corregida de *Proxy Bot* [36]. Proxy Bot implementa la arquitectura planteada, tal y como se muestra en la figura 5.3.

Proxy Bot define una interfaz Java de comunicación con StarCraft, pudiéndose implementar una IA cualquiera (*IA Usuario* en la figura) en Java que, través del Proxy Bot, permita leer el estado de juego de StarCraft así como alterarlo. Proxy Bot se comunica a través de un socket con una DLL implementada mediante BWAPI, leyendo el estado del mundo que ésta le proporciona, y enviándole a ésta las acciones que se desean ejecutar en el juego.

5.3. Diseño de los Experimentos

Crear una IA que sea capaz de jugar a un nivel competente en un juego comercial es una ardua y ambiciosa tarea. Al elegir StarCraft como entorno de pruebas se pretende demostrar que la arquitectura descrita (combinación de Darmok con árboles de comportamiento) puede dar buenos resultados en entornos reales que van más allá de entornos artificiales controlados.

La arquitectura propuesta parte de la necesidad de un sistema Darmok entrenado para competir, *a un nivel aceptable*, en el juego en cuestión. Es por ello que es necesario definir un dominio lo suficientemente detallado como para que Darmok pueda jugar correctamente a StarCraft. Una vez se ha dado una definición de dominio para Darmok y se han creado trazas de juego para entrenarlo, Darmok puede ser extendido haciendo uso de árboles de comportamiento.

5.3.1. Definición del Dominio

La definición del fichero que contiene toda la información del dominio (StarCraft) para que Darmok sea capaz de jugar a éste[1], es compleja, ya que StarCraft es un juego de estrategia comercial caracterizado por una gran complejidad en lo que respecta a los tipos de entidades disponibles, las acciones que pueden realizar, y cómo interaccionan con el resto del mundo.

StarCraft dispone de 43 tipos de unidades de combate y 45 tipos de edificios, repartidos entre las tres razas disponibles (Terran, Zerg y Protoss), y con

[1]Llevada a cabo por Santiago Ontañón.

características únicas. Cada unidad dispone de una serie de acciones básicas, a saber, *atacar*, *moverse* y *patrullar*. Ciertas unidades disponen de habilidades específicas, como es el caso de las unidades de tipo trabajador, que disponen de una acción para *recolectar minerales* y otra para *recolectar gas*. En general, muchas unidades disponen de habilidades ofensivas y defensivas de diversa índole. En su conjunto, todo ello conlleva un gran esfuerzo en la definición del fichero del dominio de StarCraft.

Por otro lado, la obtención de trazas de juego se antoja complicada, debido a que la API de StarCraft (BWAPI) no permite acceder directamente a las acciones que el jugador envía a las unidades en cada instante. Al tratarse de un videojuego comercial cerrado sin acceso a su código fuente, además, se hace imposible obtener dicha información de primera mano. Las órdenes deben inferirse de manera artificial mediante el análisis de los ficheros de las repeticiones de las partidas, a pesar de lo cual el procedimiento [2] no es del todo exacto y está sujeto a errores.

Por todo ello, y dado lo abrumador de la tarea, finalmente se creó un fichero de dominio para un aprendizaje parcial (no preparado para acabar partidas completas) con la raza Terran.

5.3.2. Experimentos

Dado que la complejidad del dominio hizo imposible preparar escenarios más elaborados, como partidas completas, la experimentación propuesta lleva a cabo un estudio del rendimiento de Darmok en escenarios de bajo nivel. Sólo se llegó a implementar la capa táctica para la gestión de acciones de bajo nivel (sección 3.1), y tampoco se llegó a desarrollar la recuperación de árboles mediante métrica de similitud basándose en el estado del mundo (sección 3.4).

A lo largo de una partida de StarCraft, Darmok tiene que hacer frente a numerosas situaciones que requieren un comportamiento altamente reactivo si se desea obtener un buen resultado. En general, cualquier forma de combate directo o uso de habilidades especiales son un buen ejemplo de ello. El que, en un combate, Darmok sea capaz de reaccionar adecuadamente, puede suponer la diferencia entre ganar o perder una batalla. El hacer un uso correcto de las habilidades especiales de ciertas unidades puede además tener como consecuencia un mejor desempeño de las labores tanto ofensivas como defensivas de la IA.

La experimentación llevada a cabo demuestra cómo el uso de árboles de comportamiento a través de la capa táctica de bajo nivel sin recuperación basada en similitud consigue mejorar sensiblemente el rendimiento de Darmok en diversos escenarios.

A grandes rasgos, los experimentos propuestos comprueban la capacidad de Darmok de reaccionar correctamente durante un combate, bien empleando las acciones básicas de las unidades enfrentadas, bien haciendo uso de sus habilidades especiales. Se proponen escenarios donde se pone a prueba distintas habilidades así como la capacidad estratégica de Darmok a la hora de emplearlas.

[2] También implementado por Santiago Ontañón.

5.4. Experimento 1

En este experimento se pone a prueba la capacidad de reacción de Darmok en un combate simple entre grupos de soldados Terran. El *soldado* es la unidad ofensiva más básica de los Terran, apenas tiene capacidad de ataque y de defensa, y puede atacar a distancia con su rifle.

Cuando Darmok analiza las trazas del juego es capaz de aprender que, en determinadas circunstancias, las unidades aliadas deben atacar a las unidades enemigas. Sin embargo, dado que Darmok no tiene una capacidad reactiva alta, es de esperar que en muchas ocasiones no reaccione ante la presencia de enemigos distintos a los marcados como objetivos del ataque, y que insista en atacar al enemigo inicialmente marcado a pesar de que pueda implicar no defenderse.

Se espera que Darmok tenga problemas a la hora de defenderse de los ataques enemigos. Si un soldado A controlado por Darmok fija como objetivo de su ataque a un soldado B enemigo, y antes de poder atacar a B, otro enemigo C aparece en su camino, es probable que C ataque a A sin que A se defienda, produciéndose una baja innecesaria. Si este comportamiento se repite para todos los soldados, es probable que el resultado global del combate se vea afectado. Para remediarlo, proponemos el uso del árbol de comportamiento de la figura 5.4. Dicho árbol comprueba continuamente, mediante una *lista dinámica de prioridad*, si hay algún enemigo visible y si el objetivo inicial al que se debe atacar (*objetivoInicial*) no es alcanzable (condición *¿SituaciónDePeligro? && NoAlcanzable(objetivoInicial)*). Si lo hay, entonces el soldado está cerca de un enemigo distinto al objetivo inicial del ataque, ante cuya presencia debe reaccionar; el enemigo es almacenado por la condición *¿SituaciónDePeligro?* en el contexto de ejecución, en la variable de nombre *enemigo*, para que, a través de la acción *Atacar* subsiguiente, sea atacado. Si no hay ninguna situación de peligro, a la unidad se le envía la orden de atacar al objetivo original, que se supone que se encuentra en una variable de nombre *objetivoInicial* del contexto. Este comportamiento se repite hasta que la acción *Atacar(objetivoInicial)* se dé por concluida, es decir, cuando el enemigo sea destruido. Cuando el enemigo es destruido, la acción *RealizarInterrupción* termina la ejecución del nodo *Interruptor*, haciendo que el árbol finalice con ello. Cuando Darmok genera una acción de tipo *Atacar*, la capa táctica crea un árbol de este tipo, e inicializa su contexto para que contenga, en la variable *objetivoInicial*, el objetivo al que se debe atacar (extraído de la acción *Atacar* generada por Darmok).

En el dominio de StarCraft hay también presentes acciones de tipo *Moverse*. Esta acción ordena a una unidad desplazarse a una determinada posición, sin prestar atención a los enemigos que pudiera encontrarse a lo largo del camino (pudiendo ser atacada por ellos de forma indiscriminada). Para solventar el carácter poco reactivo de la acción *Moverse*, se propone hacer uso del árbol de comportamiento de la figura 5.5. Este árbol muestra una estructura análoga a la del árbol que controla la acción *Atacar*. En este caso, sin embargo, cuando se detecta peligro, se reutiliza el árbol *Atacar* definido anteriormente. Si bien se podría haber hecho uso de la acción *Atacar* directamente, una de las ventajas de los árboles de comportamiento es su capacidad de reutilización. Hay que tener en cuenta que, para ciertos tipos de unidades, la acción *Atacar* podría tener que gestionarse de una manera muy distinta. Mediante la reutilización se evitaría tener que replicar el árbol original de la acción *Atacar*.

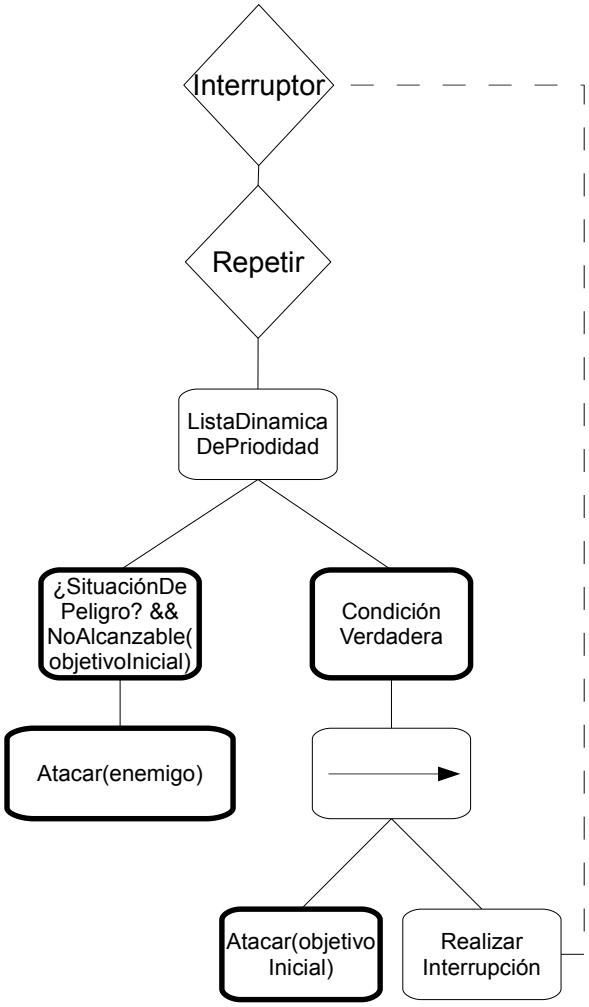

Figura 5.4: Árbol de comportamiento para la acción *Atacar*. Con un borde negro remarcado se señalan las acciones y condiciones dependientes del dominio.

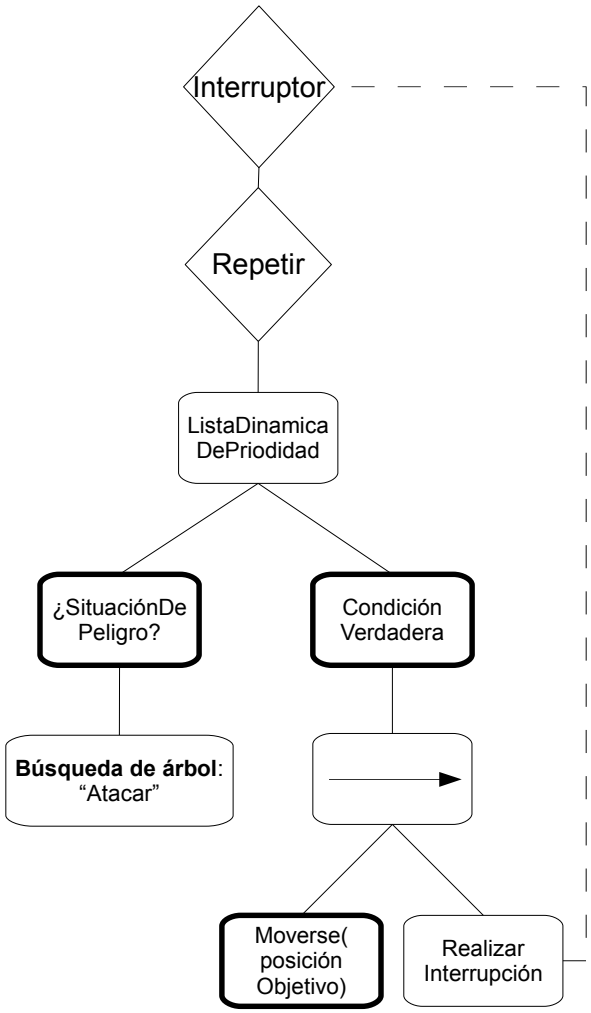

Figura 5.5: Árbol de comportamiento para la acción *Moverse*. Con un borde negro remarcado se señalan las acciones y condiciones dependientes del dominio

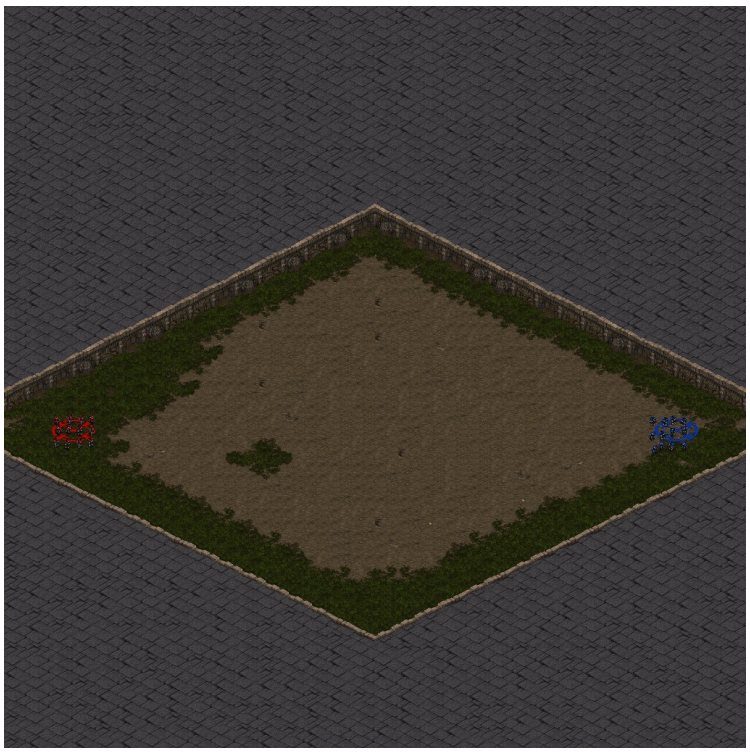

Figura 5.6: Escenario del experimento 1

El mapa de juego donde se plantea la batalla del experimento (figura 5.6) consiste en una porción de tierra plana con forma de rombo. El terreno es completamente visible a los soldados, de modo que en todo instante cada jugador sabe dónde está su oponente.

Para entrenar a Darmok se ha hecho uso de tres trazas consistentes en tres demostraciones de cómo llevar a cabo un combate en el escenario en cuestión.

El experimento a realizar consiste en la repetición de 1000 combates entre dos grupos de soldados Terran (cada grupo con 12 soldados). Por un lado se enfrenta a Darmok, sin la capa táctica de árboles de comportamiento, contra la IA estándar de StarCraft. Luego se repite el mismo enfrentamiento, pero añadiendo a Darmok la capa táctica con los árboles de comportamiento mencionados. En cada caso se mide el número de victorias sobre el total, con el fin de comparar cuál de los dos sistemas muestra un mejor rendimiento.

	Darmok sin BTs	Darmok con BTs
Victorias	262	423
Derrotas	738	577
% Victorias	26.2 %	42.3 %
% Derrotas	73.8 %	57.7 %
% Mejora	-	61.94 %

Tabla 5.1: Resultados del experimento 1

La tabla 5.1 muestra los resultados obtenidos del experimento. Se aprecia que la inclusión de los árboles de comportamiento descritos anteriormente para la gestión de las acciones *Atacar* y *Moverse* proporciona una mejora sensible de los resultados obtenidos por el Darmok estándar. En concreto, se obtiene

una mejora del 61.94 %. En este caso jugó un papel relevante el hecho de poder reaccionar ante enemigos no previstos. Analizando el desarrollo de la partida, se pudo observar que, cuando era Darmok (sin árboles de comportamiento) el que controlaba a los soldados Terran, muchos de ellos eran destruidos por el enemigo antes de que incluso éstos pudieran atacar una sola vez, debido al hecho comentado con anterioridad: los soldados no eran capaces de reaccionar ante enemigos no planificados por Darmok, y por tanto eran destruidos en el camino.

5.5. Experimento 2

En este experimento se pretende refinar la concepción que Darmok tiene del entorno de juego, para mejorar su reacción ante situaciones diversas.

La raza Terran dispone de un edificio llamado *búnker*. El búnker es un edificio protector en el cual pueden introducirse cuatro soldados, de modo que, una vez dentro, no reciben daño directo de las unidades externas. En su lugar, el búnker es el que recibe los ataques. Desde dentro del búnker, los soldados pueden atacar al enemigo, y también disponen de un mayor rango de ataque. Una vez el búnker es destruido, los soldados salen ilesos de él.

En general, es buena idea que, ante el peligro, los soldados Terran se introduzcan en los búnkeres cercanos. Así, no sólo vivirán durante más tiempo, sino que además podrán alcanzar a enemigos más lejanos. Salvo que Darmok dé una orden explícita respecto a los soldados para que se introduzcan en búnkeres, éstos no lo harán. Así, puede darse el caso de que, aunque Darmok ordene a ciertos soldados atacar al enemigo, y a pesar de que haya búnkeres cerca, su falta de capacidad para analizar de forma precisa el entorno le impida ver que dichos búnkeres podrían ser usados. Para remediarlo, se propone modificar el árbol de comportamiento de la acción *Atacar*, extendiendo el planteado en el experimento 1. La idea consiste en hacer que, si un soldado se encuentra en peligro (hay algún enemigo próximo) y hay un búnker cercano, en lugar de atacar al enemigo, decida introducirse en el búnker.

El árbol de comportamiento de la figura 5.7 recoge el comportamiento planteado. Cuando al soldado se le envía la orden de atacar a un determinado enemigo (que inicialmente se debe situar en la variable *objetivoInicial* del contexto), se comprueba si hay algún enemigo cercano y si el objetivo inicial del ataque (almacenado en la variable *objetivoInicial*) no es alcanzable. En ese caso, el soldado se encuentra cerca de un enemigo distinto del de la acción *Atacar* generada por Darmok, y por tanto debe reaccionar ante su presencia. La diferencia ahora es que el modo de reaccionar ante un peligro tiene en cuenta la presencia de búnkeres cercanos. Si hay algún búnker cercano al soldado, éste intentará entrar en él. Para ello se hace uso de una segunda lista de prioridad, que comprueba tres condiciones diferentes. En primer lugar, si el soldado ya está dentro de un búnker, es buena idea que permanezca dentro de él, así que se fuerza a que así sea. En segundo lugar, si el soldado no está dentro de un búnker pero tiene un búnker cercano, se le manda entrar en él. Por último, si ninguna de las dos condiciones anteriores se sostiene, el soldado atacará al enemigo que disparó esta rama del árbol, el cual habrá sido escrito por la condición *¿SituaciónDePeligro?* en la variable *enemigo* del contexto. Volviendo a la *lista dinámica de prioridad* de más alto nivel, en caso de que el objetivo inicial estuviera al alcance, se le mandaría atacarlo.

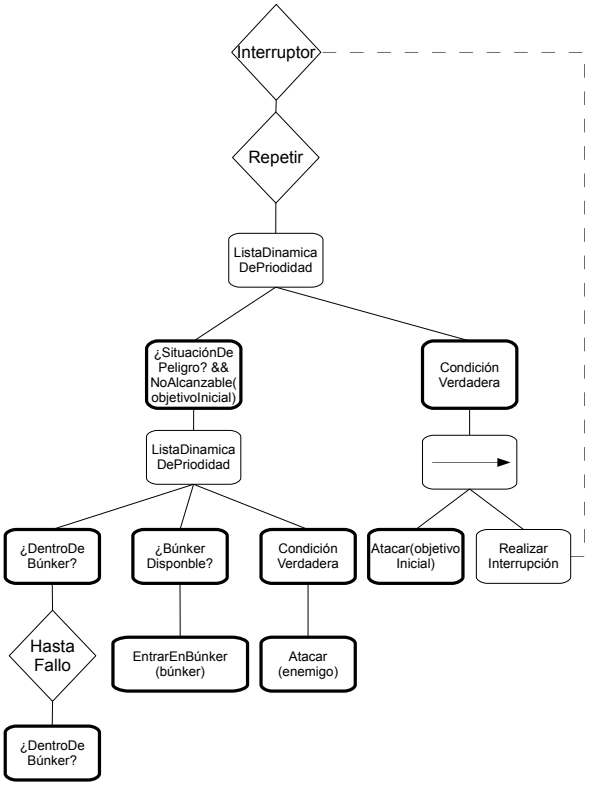

Figura 5.7: Árbol de comportamiento para la acción *Atacar*. Ahora el árbol contempla la presencia de búnkeres cercanos

Tal y como se ha explicado con anterioridad, una de las ventajas de los árboles de comportamiento es su capacidad de reutilización. Recuérdese que el árbol de comportamiento de la figura 5.5 representaba el comportamiento *moverse reactivo*. Dicho árbol reutilizaba el árbol asociado a la acción *Atacar*. Al haber actualizado el árbol de la acción *Atacar* para que tenga constancia de los búnkeres cercanos, el árbol de la acción *Moverse* también es actualizado automáticamente. Es más, cualquier otro árbol que reutilice el árbol de *Atacar* se habrá visto actualizado.

El mapa donde se ha llevado a cabo el experimento (figura 5.8) consiste en dos porciones de tierra plana, separadas por un muro. El trozo de la izquierda representa una base aliada que debe ser defendida de oleadas de enemigos. La base dispone de dos grupos de 9 soldados Terran para defenderla. En el trozo de la derecha se van generando periódicamente enemigos que se lanzan al ataque de la base aliada. En concreto, se generan 6 *zerlings* Zerg y 6 *hidraliscos* Zerg. Los zerlings son pequeñas unidades que atacan cuerpo a cuerpo (es decir, no pueden atacar a distancia), pero son débiles en general, con lo que se suelen utilizar como defensa de unidades más potentes. Los hidraliscos, por contra, atacan a distancia, infligen más daño, y son capaces de aguantar más ataques enemigos. La idea es que los zerlings defiendan a los hidraliscos mientras éstos causan el mayor daño posible. El muro que separa ambos terrenos tiene una apertura, que representa la entrada a la base. En la apertura, así como a lo largo de la base aliada, hay situados varios búnkeres, que deberían ser usados por los soldados para defenderse de los ataques.

El escenario planteado es típico en una partida de StarCraft. Es frecuente que el emplazamiento de la base aliada sea una formación geográfica con una sola entrada terrestre, estrecha. En este tipo de escenarios es una buena idea

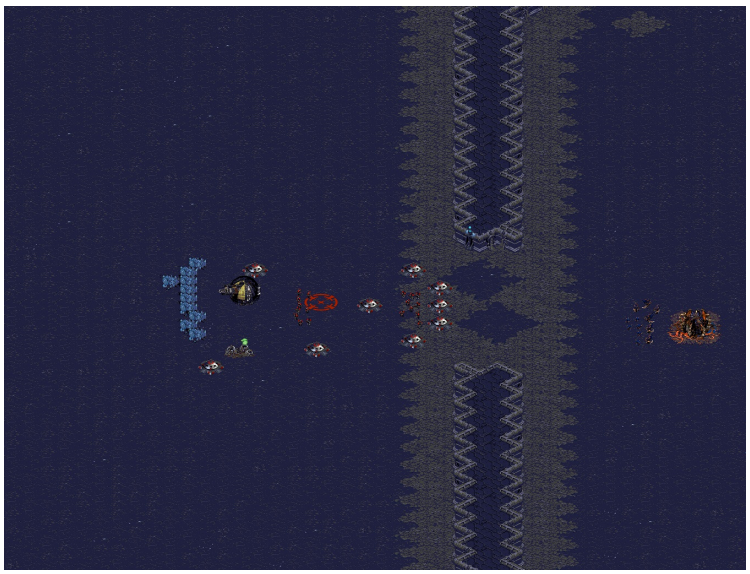

Figura 5.8: Escenario del experimento 2

concentrar las defensas en la entrada, ya que el enemigo, al menos por tierra, sólo podrá entrar por ahí.

Para entrenar a Darmok se ha hecho uso de tres trazas consistentes en tres demostraciones de cómo defenderse en el escenario mostrado. En las demostraciones, se muestra cómo los soldados deben introducirse en los búnkeres así como cómo deben atacar a los enemigos.

El experimento a realizar consiste en la repetición de 200 simulaciones de defensa de la base. En cada simulación, se envían continuamente ráfagas de 6 zerlings y 6 hidraliscos a la base aliada. Cuando la base aliada es destruida, se reinicia el escenario, y se lleva a cabo otra simulación. En cada simulación se hace un recuento de cuántas ráfagas de enemigos el jugador ha podido aguantar. En un caso, la defensa de la base la lleva a cabo Darmok sin la capa táctica de árboles de comportamiento. En el otro, es Darmok con la capa táctica de árboles de comportamiento el que lleva a cabo la defensa de la base. Con el propósito de llevar a cabo un experimento más realista, el terreno del mapa no es completamente visible al jugador que realiza la defensa. Así se evita tener una visión anticipada del ataque enemigo, lo cual va más en consonancia con una situación real, en la que el ataque puede llegar en cualquier momento sin que el jugador lo prevea.

	Darmok sin BTs	Darmok con BTs
Oleadas Resistidas	402	500
μ Oleadas por Simulación	2.01	2.5
σ Oleadas por Simulación	0.5	0.7
% Mejora	-	24.38 %

Tabla 5.2: Resultados del experimento 2

La tabla 5.2 muestra los resultados obtenidos del experimento. En total, sobre las 200 simulaciones, Darmok con la capa táctica de árboles de comportamiento es capaz de resistir 500 oleadas, mientras que sin la capa táctica

resiste 402. En promedio, Darmok sin la intervención de la capa táctica resiste 2.01 oleadas de enemigos por cada simulación, mientras que Darmok con la capa táctica resiste 2.5 oleadas.

Se deduce por tanto que incluir conocimiento experto en los árboles de comportamiento acerca del uso de búnkeres cercanos supone una mejora apreciable del rendimiento general del sistema. En particular, se obtiene un porcentaje de mejora del 24.38 %, lo cual representa casi una cuarta parte más de victorias. Al analizar la partida jugada, se pudo observar que, gracias a los árboles de comportamiento, un mayor número de soldados hizo uso de los búnkeres, con lo cual en general aguantaban más tiempo con vida, dando lugar así a la mejora de rendimiento observada.

5.6. Experimento 3

El rendimiento de ciertas unidades durante el juego depende en gran medida de cómo hagan uso de sus habilidades especiales. Un ejemplo de ello es el *buitre* Terran. El buitre es una moto de combate que inflige un daño mínimo a las unidades enemigas. A pesar de ello, dispone de una potente habilidad: la capacidad de emplazar *minas araña* en cualquier posición del mapa (en total, puede emplazar tres minas). Una mina araña es una mina que se entierra en el suelo (tras lo cual no puede ser detectada por las unidades enemigas), y que se activa cuando una unidad enemiga pasa cerca suya. Cuando la mina se activa, se acerca rápidamente a la unidad que la ha activado, y explosiona, causando un gran daño a las unidades cercanas a la explosión (es decir, la mina causa daño en toda una zona del mapa). Si a la capacidad de plantar minas araña se le añade el hecho de que el buitre Terran es la unidad más veloz del juego, el resultado es el de una unidad que, si es manejada con estrategia, puede suponer la diferencia entre una batalla ganada y una batalla perdida. Una estrategia que suele ser usada por expertos jugadores consiste en mandar buitres Terran a plantar minas araña en medio de las tropas enemigas, tras lo cual abandonan la zona rápidamente. Aunque muchas minas puedan ser interceptadas por el enemigo, basta que exploten unas pocas para que se causen daños irreversibles en el ejército contrario.

Es difícil que Darmok aprenda a usar de manera precisa las minas araña. Por un lado, cuando Darmok genera la orden *EmplazarMinaAraña*, debe de llevar un paso previo de adaptación (basada en el ciclo CBP) para calcular el nuevo lugar donde situar la mina, pudiendo ser que dicho lugar no sea el adecuado. Por otro, durante un combate Darmok debería ser consciente de que, siempre que pudiera, debería hacer uso de minas araña, y que además debería de huir de ellas si hubiera enemigos cerca, puesto que de lo contrario la explosión podría alcanzar a las unidades aliadas.

Para solventar estos inconvenientes, se propone implementar un árbol de comportamiento para el buitre Terran (figura 5.9), que se encargue de gestionar su acción *Atacar*. Este árbol muestra una estructura similar al del soldado Terran del experimento 1 (sección 5.4), es decir, es un árbol reactivo ante la presencia de enemigos cercanos. Sin embargo, ahora, cuando el buitre detecte la presencia de enemigos y compruebe que dispone de minas araña, intentará emplazarlas en una posición cercana a los enemigos detectados. Además, si el buitre detecta que en las proximidades hay tanto enemigos como minas araña, intentará huir de la zona, ante el riesgo de que las minas hagan ex-

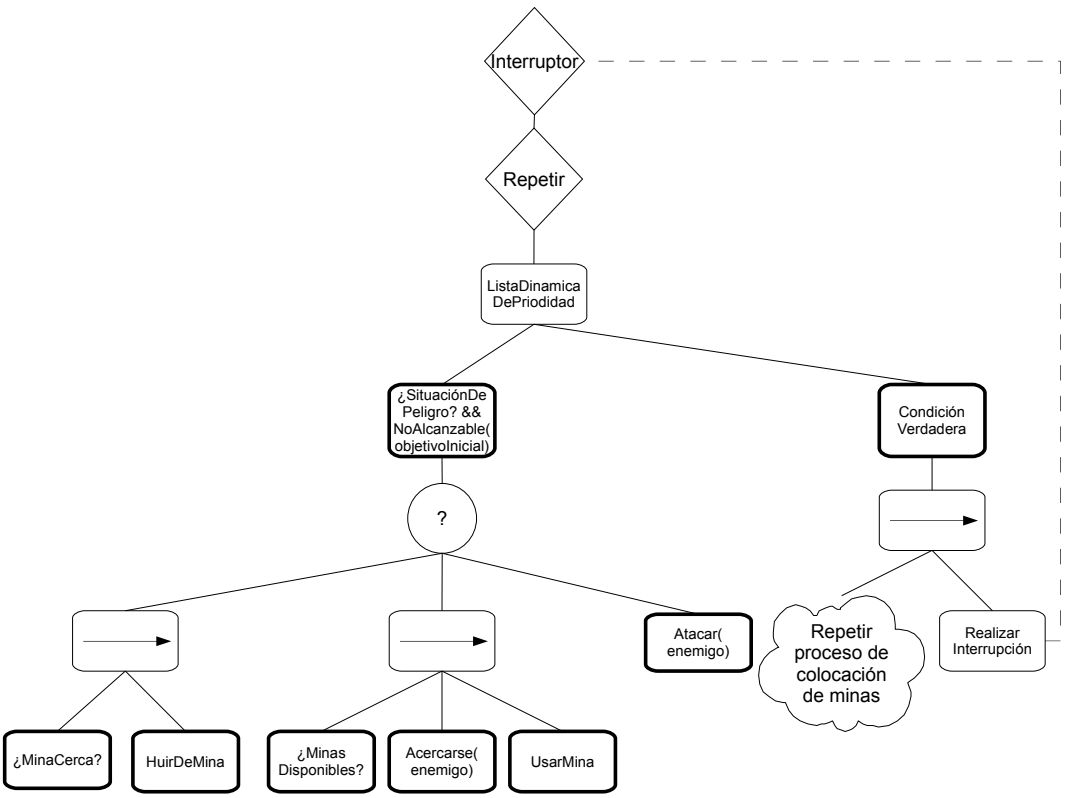

Figura 5.9: Árbol de comportamiento para la acción *Atacar* del buitre Terran

plosión y le hagan daño en el proceso. En otro caso, si no dispone de minas araña, atacará al enemigo mediante un ataque normal. Si el objetivo original del ataque (extraído de la acción *Atacar* generada por Darmok) es visible, intentará realizar el mismo proceso, pero contra dicho enemigo.

El árbol de comportamiento de la figura 5.9 recoge de manera simplificada el comportamiento explicado. En la implementación real, dado que la *nube* de la derecha marcada con el mensaje *repetir proceso de colocación de minas* coincide con la parte de la izquierda del árbol, se llevó acabo una abstracción de todo el proceso, para que pudiera ser reutilizado en ambos lugares sin necesidad de repetir toda la estructura.

El mapa donde se ha llevado a cabo el experimento es el de la figura 5.10. Este escenario es similar al del experimento 1. En este caso, sin embargo, se enfrentan, por un lado, un grupo de 6 buitres Terran controlados por Darmok, y por otro, un grupo de 24 *murciélagos de fuego* controlados por la IA estándar de StarCraft. Un *murciélago de fuego* es un soldado Terran cuya arma es el lanzallamas, y que sólo puede atacar, por tanto, cuerpo a cuerpo. La idea del experimento es comprobar cómo de efectivo es el proceso de colocación de minas araña de Darmok en lo que a infligir daño en el ejército enemigo se refiere. En un escenario normal, no es de esperar que los 6 buitres puedan derrotar a los 24 murciélagos de fuego, motivo por el que, en realidad, en el experimento se llevará a cabo un recuento de cuántos murciélagos de fuego han conseguido ser eliminados.

Para entrenar a Darmok se ha hecho uso de tres trazas distintas, jugadas en el escenario planteado, en las que se muestra cómo llevar a cabo el emplazamiento de minas araña.

El experimento consiste en llevar a cabo 500 repeticiones del combate explicado. Para cada combate, se calcula cuántos murciélagos de fuego han sido

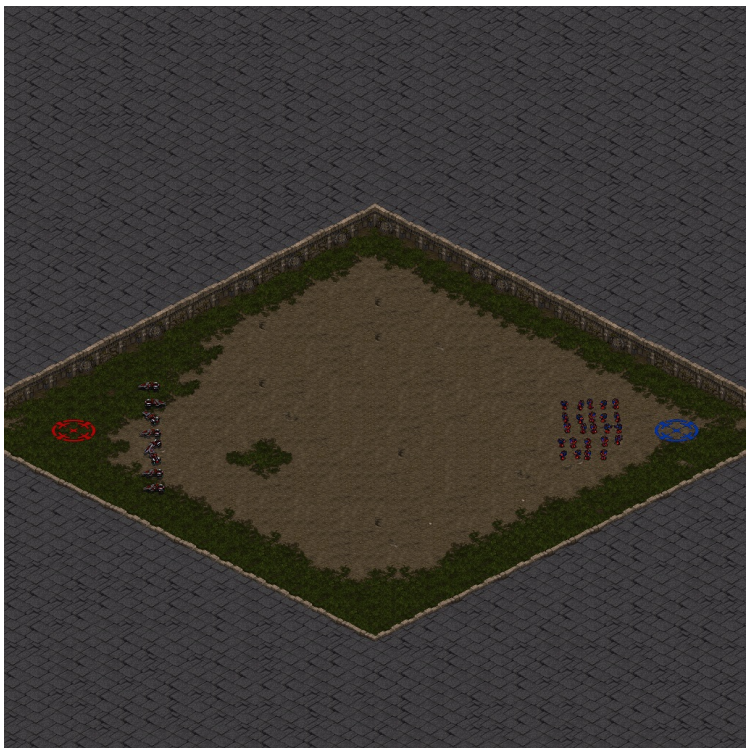

Figura 5.10: Escenario del experimento 3

eliminados, con el fin de comprobar la diferencia de rendimiento cuando es sólo Darmok el que controla al jugador al mando de los buitres, y cuando Darmok es extendido mediante la capa táctica de bajo nivel. El terreno del mapa es completamente visible a ambos jugadores, de modo que en todo momento conocen la localización exacta de las unidades enemigas. La tabla 5.3 muestra los resultados del experimento.

	Darmok sin BTs	Darmok con BTs
Murciélagos de Fuego Muertos	4927	6607
μ Murciélagos por Simulación	9.85	13.21
σ Murciélagos por Simulación	5.42	5.83
% Mejora	-	34.10 %

Tabla 5.3: Resultados del experimento 3

Nuevamente se puede comprobar que la inclusión de árboles de comportamiento para la gestión de acciones de bajo nivel supone una mejora respecto al Darmok puro. En este caso, se ha producido un incremento del 34.10 % en el número total de murciélagos de fuego destruidos, y una diferencia de 3.36 murciélagos de fuego en promedio por cada combate. Tras analizar el desarrollo de la partida, se pudo observar que, en numerosas ocasiones, Darmok no era capaz de situar las minas araña en posiciones adecuadas. Los árboles de comportamiento las situaban más cerca de los enemigos, con lo que causaban en general mayor daño. Además, los buitres controlados por los árboles de comportamiento se veían menos afectados por las explosiones de las minas araña, al ser consciente de su presencia y poder huir del lugar a tiempo, antes de que hiciera explosión.

Capítulo 6

Conclusiones y Trabajo Futuro

Este trabajo se ha centrado en el estudio de la idoneidad de la combinación de conocimiento experto, en forma de árboles de comportamiento, con Darmok, un sistema de planificación basada en casos para juegos de estrategia en tiempo real.

Darmok presenta varios problemas en lo que respecta a su capacidad de reacción ante eventos acontecidos en el mundo. En particular, en ciertos escenarios (en la gestión de acciones de bajo nivel) muestra una baja reactividad que le impide tomar decisiones que pueden resultar trascendentales para el transcurso de batallas; en otros escenarios (en la gestión de ciertos planes objetivo), muestra una reactividad excesivamente alta, pudiendo cancelar planes completos por la mera falla de una de sus acciones constituyentes.

Para solventar ambos problemas, hemos propuesto una arquitectura híbrida basada en la inclusión de conocimiento experto en forma de árboles de comportamiento, y gestionado mediante una capa táctica que se comunica directamente con Darmok. Para probar la validez de la arquitectura presentada se llevó a cabo un estudio experimental en un juego de estrategia en tiempo real, StarCraft. Dada la complejidad del dominio (StarCraft), sólo llegó a implementarse la capa táctica para la gestión de acciones de bajo nivel, omitiéndose detalles tales como la recuperación basada en similitud de los árboles de comportamiento. Se propusieron una serie de experimentos en los que se comprobó la diferencia de rendimiento entre Darmok sin la presencia de árboles de comportamiento y Darmok con la incorporación de árboles de comportamiento. De los resultados obtenidos en la experimentación llevada a cabo se desprende, no sólo que la inclusión de conocimiento experto en forma de árboles de comportamiento supone una mejora significativa del rendimiento de Darmok, sino que esta mejora se puede dar en escenarios de muy diversa índole.

Hasta donde llega nuestro conocimiento, no hay ningún otro trabajo que combine planificación basada en casos con árboles de comportamiento. No obstante, este enfoque puede considerarse como un ejemplo particular de una tendencia en IA más general, que busca la combinación de datos empíricos con el modelo del dominio: en este caso, los casos de la biblioteca de planes representan los datos empíricos, mientras que los árboles de comportamiento codifican una visión parcial del modelo del dominio por parte del experto. Desde este punto de vista, se puede encontrar en la literatura [27] muchos ejemplos de combinación del modelo del dominio, generalmente en forma de reglas, con razonamiento basado en casos (CRB). Algunos sistemas toman la

salida de un componente basado en reglas y la utilizan como entrada de uno basado en casos, como el descrito en [13], un sistema de auditoría bancaria que detecta automáticamente transacciones fuera de lo normal, irregulares, o arriesgadas, aplicando posteriormente CBR, el cual inspecciona las transacciones detectadas y calcula un nivel de castigo. Por otro lado, otros sistemas toman un enfoque más cercano al presentado en este trabajo, donde la salida de un módulo basado en casos alimenta a uno basado en reglas. Un ejemplo es el descrito en [18], un sistema médico para pacientes con Alzheimer, donde se llama al módulo basado en casos para determinar si se debería recetar un fármaco neuroléptico al paciente; de ser así, el sistema basado en reglas se encarga de seleccionar uno de entre cinco fármacos posibles.

Si consideramos a los árboles de comportamiento como un tipo particular de artefacto de planificación que almacena planes codificados a mano, también podemos encontrar trabajo relacionado centrado en la combinación de planificación basada en casos y otros enfoques de planificación. El sistema SiN [20] hace uso de un algoritmo de planificación basada en casos que combina recuperación de casos conversacional con planificación generativa. SiN puede generar planes dado un modelo del dominio incompleto mediante el uso de casos con los que extender dicho modelo, el cual se proporciona en forma de dominio de planificación. SiN puede además razonar con información no exacta del estado del mundo, incorporando preferencias en los casos. Mientras que en SiN el módulo basado en casos y el modelo del dominio se desarrollan independientemente el uno del otro, nosotros proponemos un enfoque más eficiente basado en desarrollar un modelo del dominio específico para cubrir los vacíos presentes en los datos empíricos (obtenidos en forma de trazas).

Respecto a posible trabajo futuro, el primer paso sería completar la implementación de la arquitectura planteada, ya que, debido a la complejidad del dominio de juego, StarCraft, no pudo finalizarse a tiempo. Por un lado, la incorporación de recuperación de árboles de comportamiento basada en un criterio de similitud puede suponer un empuje positivo en lo que al rendimiento de la arquitectura se refiere, ya que es de esperar que árboles particularizados para cada tipo de situación se comporten de mejor manera que árboles genéricos que no tienen en cuenta el estado del mundo en el que son aplicados. Por otro lado, la gestión de planes objetivo mediante árboles de comportamiento podría mejorar el rendimiento en aquellos objetivos para los que Darmok no ha conseguido aprender planes lo suficientemente estructurados.

También contemplamos la exploración de posibles técnicas para facilitar la tarea de identificar las áreas de la biblioteca de planes que requieren la intervención del experto. Tenemos en mente la idea de un proceso de identificación asistido por computador en el que se generen trazas de la IA gestionada por la biblioteca de planes, para que el sistema automáticamente detecte aquellos objetivos y acciones que fallan con cierta frecuencia como sitios donde se deba llevar a cabo una posible mejora.

Apéndice A

JBT, Guía del Usuario

A.1. Introduction

Java Behaviour Trees (JBT) is a Java framework for building and running behaviour trees (BTs). In the past few years, BTs have been widely accepted as a tool for defining the behaviour of video games characters. However, to the best of our knowledge, there is no free-software Java implementation of such technology. With JBT we intend to provide a solid framework to build and run BTs in Java.

JBT has two main parts. On the one hand, there is the JBT Core (it is the Eclipse SDK project under the "./JBTCore"directory of the repository), which implements all the classes needed to create and run BTs. JBT Core basically lets the user create BTs in pure Java and then run them. In order to ease the task of creating BTs, JBT Core includes several tools that automatize the process of creating BTs. In particular, it can create the Java source code of a BT from its description in an XML format. By doing so, the user of this framework basically has to worry only about defining BTs in XML files and implementing the low level actions and conditions that his trees will use, which are domain-dependant (that is, they depend on the game being played). We provide a .jar file with all the JBT Core classes. Of course, in order to get the last version of the JBT Core the repository can be accessed.

On the other hand, there is the JBT Editor (which is composed of two Eclipse SDK projects under the "./JBTEditor"directory of the repository). The JBT Editor is a GUI application that can be used for defining BTs, and then exporting them into XML files in the format that the JBT Core understands. The JBT Editor offers a set of standard nodes[1] for building BTs. It includes nodes such as sequences, parallels, decorators, etc. For low level actions and conditions, the user can provide their conceptual definition through Make Me Play Me (MMPM) domain files (for more information on MMPM, see the Sourceforge page of the project "Darmok 2"). The JBT Editor is an Eclipse RCP application, so you must use Eclipse SDK in order to run it. An alternative to run it is to use the executable files provided for each platform. Of course, if order to get the last version of the JBT Editor the repository can be accessed.

JBT implements a BT model which is mainly based on that of the book .Artificial Intelligence for Games", second edition, by Ian Millington and John Funge. JBT also includes the concept of "guard.and static and dynamic priority

[1]Note that, when talking about BTs, *node* and *task* are used interchangeably.

lists, which make use of guards. JBT BTs are driven by ticks, which means that, in order for them to have CPU time, they need to be externally ticked. By following this pattern, the user can control how much CPU time the BT consumes.

In this document we explain how JBT can be used to build and run BTs. This process has the following steps:

- Defining low level actions and conditions to be used in the trees. These actions and conditions are defined in the MMPM format.
- Implementing the low level actions and conditions. The user has to define how the low level actions and conditions work. JBT does not know how these domain-dependent actions and conditions work, so the user has to provide a Java implementation of them.
- Creating BTs with the JBT Editor. Here, the user creates BTs that are exported into generic XML files.
- Creating the Java declaration of the BTs that were declared in the XML files. This is automatically done by one of JBT's tools.
- Running the BTs by using the core classes of JBT.

In the next sections we will describe all of these steps. Also, we will conceptualize them through a real example on a real game, since we will build a tree that is able to control a Terran Marine of the Real Time Strategy Game StarCraft.

A.2. JBT, an Overview

In this section we describe the JBT architecture as well as the main features that BTs have.

A.2.1. Model Driven by Ticks

JBT implements a BT model driven by ticks. A BT must be evaluated through ticks, so every game cycle an external caller *ticks* the tree in order for the tree to update its status. A tick is just a way of giving the tree some CPU time to update their status; in particular, ticks are used to give the nodes of the tree some time to evaluate whether they have finished or not, and consequently make the tree evolve.

The simplest approach to BTs driven by ticks is that of ticking the root node and then letting each node recursively tick its children according to its semantics. However, this is a very inefficient process, since in general the major part of the nodes of the tree are just waiting for their children to finish. Therefore, they should not receive ticks, since unless their children are done they will do nothing useful when receiving the tick. Therefore, in general only very few nodes should be ticked at a game cycle, and as a result JBT implements a model in which there is a list of *tickable* nodes. Only the nodes in the list can be ticked.

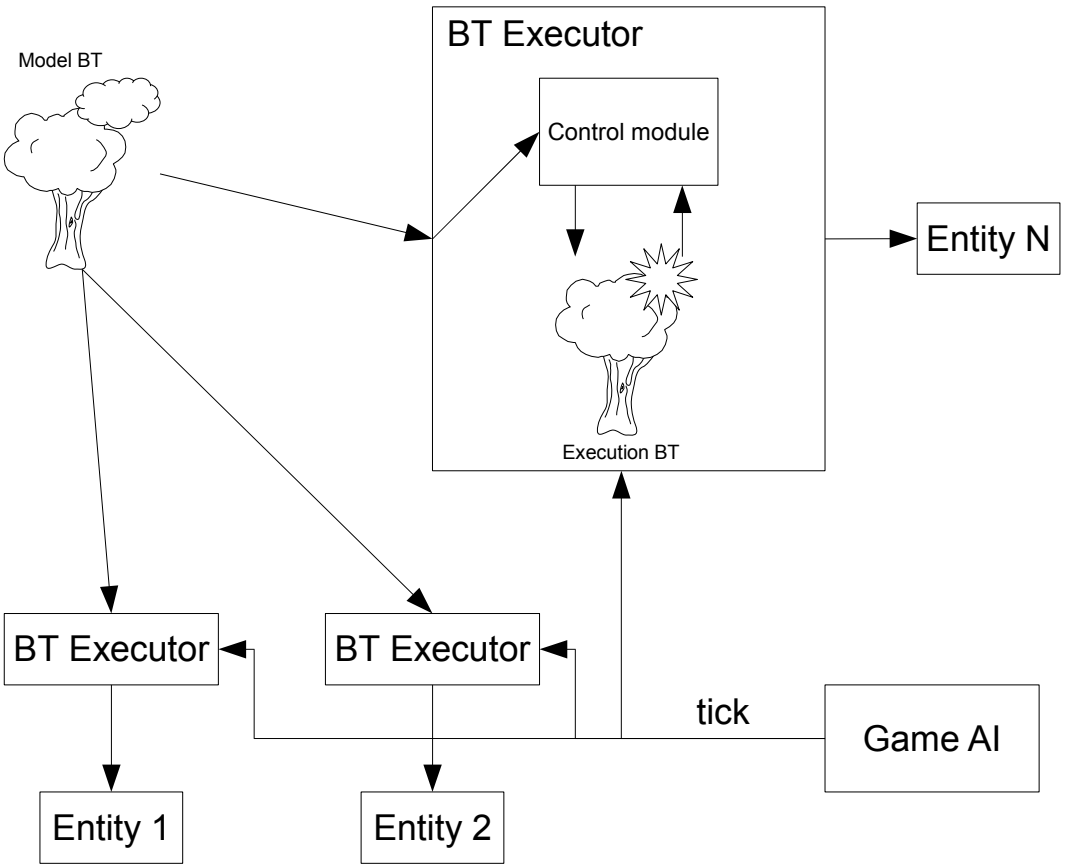

Figura A.1: Overview of the BT architecture

A.2.2. Model Independent from Execution

When running a BT, there should be a clear distinction between the tree that is being run (the model) and how it is actually being run (the execution). For each particular behaviour, we distinguish between the *Model BT* that defines it and how it is being run. The *how* is what the *BT Executor* does. Basically, for every entity in the game that wants to run a behaviour (Model BT), there is a BT Executor. The BT Executor takes the Model BT and processes it (without modifying it), simulating the behaviour that is represented by the Model BT. This choice implies that, apart from the Model BT, there is another type of tree, the *Execution BT*. When an entity wants to execute a behaviour, the BT Executor takes the Model BT and creates an Execution BT to execute the behaviour. The BT Executor along with the Execution BT know how to run the behaviour that the Model BT represents.

A.2.3. Architecture

Figure A.1 shows an overview of the JBT Core architecture. There is a Model BT that represents a particular behaviour. Also, there is a BT Executor for every entity that wants to run the Model BT. Each BT Executor makes use of the Model BT and builds an Execution BT that actually runs the behaviour conceptualized by the Model BT. An external Game AI ticks the BT Executors, in order for them to update the trees that they are running.

The user of the framework does not have to know all the details about how JBT internally works. However, since he has to implement some classes

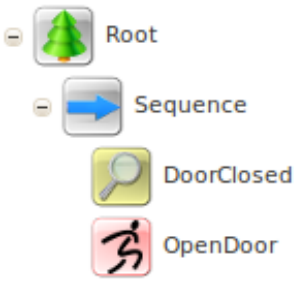

Figura A.2: a simple behaviour tree

in order to run his own trees, at least he should know the general architecture of JBT.

A.2.4. BT Model

Before even starting to explain all the steps required to build and run BTs with JBT, we have to first think about what BT model JBT offers. JBT implements a BT model that is mainly based on that of [19]. Our model also include guards and static and dynamic priority lists, as described in [8]. With this model the user can implement a wide range of behaviours.

For instance, the tree of figure A.2 represents a simple tree that is used by a game character that wants to open a door. First of all, it checks if the door is closed (condition *DoorClosed*). If so, then it tries to open it by executing the action *OpenDoor*.

In the tree of figure A.2 we can see four nodes. The node called *Root* is just the root of the tree, and it has no actual meaning apart from it. Then there is a *Sequence* node, which runs in sequence both of its children, the *DoorClosed* condition and the *OpenDoor* action. The Sequence node is a standard node, but both the DoorClosed and the OpenDoor nodes are domain dependent, that is, they have been defined by the user so they have a useful meaning within the context of the game being played.

The tree of figure A.3 represents the behaviour of a character that is trying to enter a room. The topmost selector succeeds as long as one of its children succeed. The first child tries to enter the room when the door is locked. In such case, the character tries several tactics to open the door. First, if it has the key, it uses it to open the door. If it does not have the key, but it has a grenade, then it uses the grenade in order to blow the door up. Finally, if none of the above conditions are met, the character will try to enter the room through its window (note that here a *Subtree Lookup* node is used. This node just runs a tree that is already defined; in this case, the tree that will be run is *EnterThroughWindow*). On the other hand, if the door is not locked and it is closed, the character will just open it up.

A.2.4.1. Execution Context

All nodes in a BT have an *execution context*, which is usually shared by all of them. The execution context, or *context* for short, acts as a blackboard that can be used by nodes in order to write and read variables. For instance, a node may write a variable into the context, under a name *MyVariable*. This variable can be read then by using its name, *MyVariable*. This way, the context can be

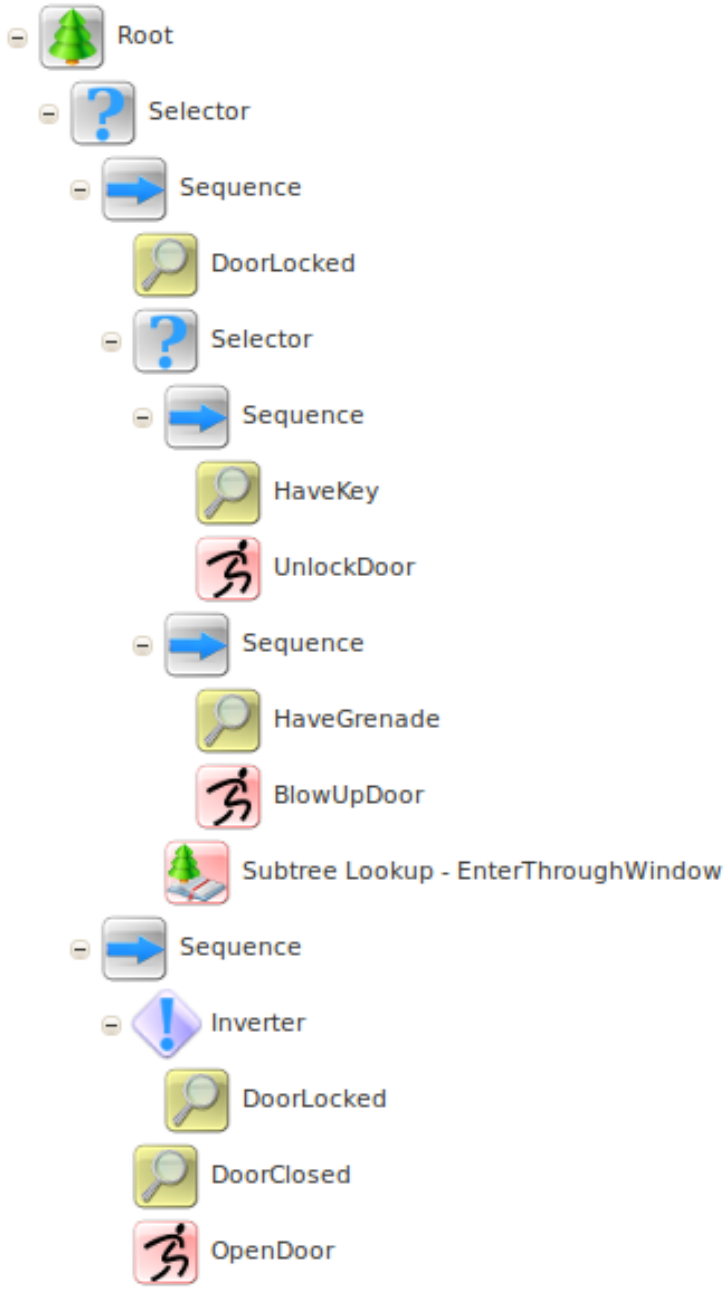

Figura A.3: a complex behaviour tree

seen as a way for the nodes of a BT to communicate.

However, not always all the nodes share the same context. In general, the context of a BT is passed down from parents to children. Thus, the initial context is that of the root of the tree, which will pass it to its children. The root's children will pass the context to their own children, and so on. Nevertheless, some nodes do not pass their own context to their children, but another one instead. This new context may be empty or not, and it may be of a different type. JBT supports the following types:

- Basic Context: this is just a normal context, with no especial features. It is the context that the user of the framework can create. Other types of contexts are managed through decorator tasks, and the user cannot create them.

- Hierarchical Context: a Hierarchical Context has another context (the *input context*) as a base. When the Hierarchical Context cannot find a variable within its own set of variables, it will ask its input context for the variable. Note that the Hierarchical Context can be used to build a complex hierarchy of context: if the input context is a Hierarchical Context too, the request for the variable may go up the hierarchy until a non-Hierarchical Context is reached.

- Safe Context: a Safe Context has another context (the *input context*) as a base. Initially, all variables are read form the input context. However, when a variable is modified, its value is not modified in the input context, but locally modified instead. From then on, the variable will be locally read (that is, from the set of variables of the Safe Context) instead of reading if from the input context. Thus, the input context is never modified. A Safe Context can be used to situations in which a certain context (the input context) must be used in read-only mode.

- Safe Output Context: a Safe Output Context behaves much in the same way as the Safe Context. It has another context, the *input context*, as a base. However, this context also contains a set of *output variables* (that is, a list of variablesñames). The list of output variables represents the variables that can be modified in the input context. Variables other than those in the list of output variables will be stored locally in the set of variables of Safe Output Context, just as if it were a Safe Context. Thus, when the Safe Output Context modifies the value of a variable, it will normally set its value in a local variable (that is, a variable belonging to the Safe Output Context). However, if the variable is one of the list of output variables, the value will be set in the input context, which will therefore be modified. When retrieving variables, a variable in the list of output variables will always be retrieved from the input context. A variable that is not in the list of output variables will also be retrieved from the input context; however, when such variable is modified, the value will be retrieved from the Safe Output Context (that is, from the moment a variable that is not in the list of output variables is modified, it is managed locally).

A.2.4.2. Native Tasks

JBT offers a wide range of tasks that can be used to build behaviour trees. JBT basically implements the BT model described in [19], but extended with guards.

JBT supports the following tasks:

- Composite tasks: tasks with one or more children, whose execution depends on the execution of their children. The task's children are ordered.
  - Sequence: task that sequentially executes all its children in order. If one fails, the Sequence task fails. If all succeeds, the Sequence task succeeds.
  - Selector: task that sequentially executes all its children in order. If one succeeds, the Selector task succeeds. If all fail, the Selector task fails.
  - Parallel: task that concurrently executes all its children. A Parallel task does have a *parallel policy*. If the parallel task's policy is *sequence*, the parallel fails if one child fails; if all succeed, then the parallel succeed. If the parallel task's policy is *selector*, the parallel fails if all its children fail. If one succeeds, then the parallel also succeeds.
  - Random Selector: task that executes all its children in a random order. If one fails, the Sequence task fails. If all succeeds, the Sequence task succeeds.
  - Random Sequence: task that sequentially executes all its children in random order. If one succeeds, the Selector task succeeds. If all fail, the Selector task fails.
  - Dynamic Priority List: task that executes the child with the highest priority whose guard is evaluated to true. At every AI cycle, the children's guards are re-evaluated, so if the guard of the running child is evaluated to false, it is terminated, and the child with the highest priority starts running. The Dynamic Priority List task finishes when no guard is evaluated to true (thus failing) or when its active child finishes (returning the active child's termination status).
  - Static Priority List: task that executes the child with the highest priority whose guard is evaluated to true. Unlike the Dynamic Priority List, the Static Priority List does not keep evaluating its children's guards once a child is spawned. The Static Priority List task finishes when no guard is evaluated to true (thus failing) or when its active child finishes (returning the active child's termination status).
- Decorator tasks: tasks with one child whose purpose is to alter the way other tasks behave.
  - Interrupter: task that controls the termination of its child task. An Interrupter simply lets its child task run normally. If the child returns a result, the Interrupter will return it. However, the Interrupter can be asked to terminate the child task and return an specified

status when done so. The task that can interrupt an Interrupter is the Perform Interruption task.

- Inverter: task used to invert the status code returned by its child. When the decorated task finishes, its status code gets inverted.
- Limit: task that limits the number of times a task can be executed. This decorator is used when a task (the child of the decorator) must be run a maximum number of times. When the maximum number of times is exceeded, the decorator will fail forever on.
- Repeat: task that runs its child task forever. When its child task finishes, it runs it once more.
- Until Fail: task that runs its child as long as it does not fail. When the child task fails, Until Fail succeeds.
- Hierarchical Context Manager: task that creates a new context for its child. The context that it creates is an empty (with no variables) Hierarchical Context whose input context is the context that is passed to the Hierarchical Context Manager.
- Safe Output Context Manager: task that creates a new context for its child. The context that it creates is an empty (with no variables) Safe Output Context whose input context is the context that is passed to the Safe Output Context Manager.
- Safe Context Manager: task that creates a new context for its child. The context that it creates is an empty (with no variables) Safe Context whose input context is the context that is passed to the Safe Context Manager.

- Leaf tasks: tasks with no children.

  - Wait: task that keeps running for a period of time, and then succeeds. The user can specify for how long (in milliseconds) the Wait task should be running.
  - Subtree Lookup: see the following sections to see what this node does.
  - Perform Interruption: task that interrupts an Interrupter task.
  - Variable Renamer: task that renames a variable in the context.
  - Success: task that immediately succeeds.
  - Failure: task that immediately fails.
  - Action: generic action that is executed in the game engine.
  - Condition: generic condition that is executed in the game engine.

A.3. Step 1: Defining Low Level Actions and Conditions

The first step[2] when creating BTs is to define the set of low level actions and conditions that the trees will be using. These actions and conditions are

[2]Well, it does not necessarily have to be the first step, but we have to start at somewhere.

domain dependent, that is, they depend on the game that the trees will be run for. For instance, if we are dealing with a first person shooter (FPS from now on), then we may need actions and conditions such as those used in the trees of section A.2.4.

However, we are going to build a more complex example. Here we are going to define a behaviour tree that is able to control a Terran Marine of StarCraft, so we have to define actions and conditions that are useful for such context. The behaviour that we want to implant in the Terran Marine is as follows:

The marine is constantly checking three conditions. If there is no danger around the marine, then he just patrols around its current position. Patrolling around a position means that the marine will move randomly around a central point, and will attack whatever he finds on its way. However, if he finds himself in a low level danger situation (that is, a dangerous situation he thinks he can survive), he will try to kill whatever enemy finds dangerous. On the other hand, if he finds himself in a high level danger situation (that is, a dangerous situation he thinks he cannot survive), he will run away to the closest base.

We therefore define some actions and conditions that will be used by the BT:

- Actions:
  - Attack: this action just makes the marine attacks a specific unit.
  - Move: this action makes the marine move to a specific target position on the map.
  - AttackMove: this action makes the marine mote to a specific target position on the map. Also, if he finds an enemy on its way, he will combat the enemy.
  - ComputeClosestBasePosition: this action computes the position of the base that is closest to the marine.
  - ComputeCharacterPosition: this action just computes the current position of the marine.
  - ComputeRandomClosePosition: given a position A, this action computes a random position that is close to A.
- Conditions:
  - LowDanger: this condition checks if the marine is in a low danger situation.
  - HighDanger: this condition checks if the marine is in a high danger situation.

Actions and conditions must be defined according to the MMPM domain file format. For those unaware of what MMPM is, this does not really pose a problem, since what we are really interested in is the format that MMPM follows in order to define actions and conditions.

A MMPM domain file defines the conceptual level of a game (its *domain*), by declaring what *entities*, *actions*, *sensors* and *goals* are present in the game. We will not describe all of them, but only what we need to declare actions and conditions that can be used in BTs.

A MMPM domain file has the following structure:

```
<Domain package="valid Java package name">
  <ActionSet>
    <!-- Actions declaration -->
  </ActionSet>

  <SensorSet>
    <!-- Sensors declaration -->
  </SensorSet>

  <GoalSet>
    <!-- Goals declaration -->
  </GoalSet>

  <EntitySet>
    <!-- Entities declaration -->
  </EntitySet>
</Domain>
```

However, we are only interested in the set of actions and conditions, so $< GoalSet >$ and $< EntitySet >$ can be left empty (but they actually have to be present). The $< ActionSet >$ element defines the set of actions that are present in the game, which are also the set of low level actions that can be used when building BTs. An $< ActionSet >$ element contains a sequence of $< Action >$ elements, each one being an action. An $< Action >$ element has one attribute, its name (which is called *name*). The $< SensorSet >$ element defines the set of *sensors* that can be used in the game. A $< SensorSet >$ element contains a sequence of $< Sensor >$ elements, each one being a sensor. A $< Sensor >$ element has two attributes: its *name* and its *type*. In MMPM, a *sensor* is an operation that queries something about the world. As a result, a sensor can return *any*[3] type of value. Here we are interested in sensors whose type is *boolean* (*BOOLEAN* in the MMPM domain file), since they represent what in BTs is known as conditions, that is, a query operation that returns either true or false. Therefore, the set of boolean sensors of the MMPM domain file is the set of conditions that can be used when building BTs.

Both actions and sensors may have input parameters. An input parameter is a parameter that is supposed to be used by the action or sensor when running. For instance, the *Move* action above does have one input parameter, which is the target position where the unit must go to. An input parameter has a name and a type. Thus, both $< Action >$ and $< Sensor >$ elements may have a sequence of $< Parameter >$ elements, each one being a parameter. Each $< Parameter >$ element has two attributes, its *name* and its *type*. Therefore, actions and sensors have the following structure:

```
<Action name="MyActionName">
  <Parameter name="ParameterName1" type="ParameterType1"/>
  ...
  <Parameter name="ParameterNameN" type="ParameterTypeN"/>
</Action>
```

[3]Really not *any*.

```
<Sensor name="MySensorName" type="BOOLEAN">
  <Parameter name="ParameterName1" type="ParameterType1"/>
  ...
  <Parameter name="ParameterNameM" type="ParameterTypeM"/>
</Sensor>
```

MMPM supports the following types for parameter types: *FLOAT*, *BOOLEAN*, *STRING*, *INTEGER*, *DIRECTION*, *COORDINATE*, *PLAYER*, *ENTITY_ID* and *ENTITY_TYPE*. This set of parameter types may seem overwhelming, which is why in general several of them are not used. *FLOAT*, *BOOLEAN*, *STRING* and *INTEGER* are self-explanatory. *DIRECTION* represents an integer value, which is why, if it is used as the type of a parameter, JBT will treat it just as an integer. *COORDINATE* represents a coordinate in an N-dimensional coordinate system. In practice, a *COORDINATE* value is a non-empty sequence of real values (for instance, "23 -4.5 67", 3.45 .ºr "12.45 -0.34 9.44 -12.3"). *PLAYER* represents the name of a player of the game, so in practice it is treated as a string (*STRING*). *ENTITY_ID* represents the identifier of an entity in the game. In practice, it is treated as a string (*STRING*). *ENTITY_TYPE* represents the type of an entity. In practice, it is treated as a string (*STRING*).

As a consequence, the user will generally use just *FLOAT*, *BOOLEAN*, *STRING*, *INTEGER* and *COORDINATE*, since the rest of MMPM parameter types are equivalent to *STRING*.

MMPM format, however, does not include an important parameter type, *object*. In general, there will be actions and sensors will make use of input parameters of many types. In order to be able to manage a wide range of types, JBT extends the MMPM domain file format so that parameters also accept the *OBJECT* type. An *OBJECT* is just a variable of any type.

The MMPM domain file that defines the set of actions and conditions for the Terran Marine example is as follows:

```
<Domain package="mypackage">
  <ActionSet>
    <!-- Orders the current unit to attack another unit -->
    <Action  name="Attack">
      <Parameter name="target" type="ENTITY\_ID"/>
    </Action>

    <!-- Orders the current unit to go to a target
    position -->
    <Action  name="Move">
      <Parameter name="target" type="COORDINATE"/>
    </Action>

    <!-- Orders the current unit to go to a target position.
    If an enemy is found along the way, the unit will
    combat him -->
    <Action name="AttackMove">
      <Parameter name="target" type="COORDINATE"/>
    </Action>
```

```
    <!-- Orders the position of the base that is closest to
    the current unit -->
    <Action  name="ComputeClosestBasePosition"/>

    <!-- Computes the position of the current unit -->
    <Action  name="ComputeCharacterPosition"/>

    <!-- Computes a random position that is close to the
    input position -->
    <Action name="ComputeRandomClosePosition">
      <Parameter name="initialPosition" type="COORDINATE"/>
    </Action>
  </ActionSet>

  <SensorSet>
    <!-- Checks if the current unit is in a low danger
    situation -->
    <Sensor name="LowDanger" type="BOOLEAN"/>

    <!-- Checks if the current unit is in a high danger
    situation -->
    <Sensor name="HighDanger" type="BOOLEAN"/>
  </SensorSet>

  <EntitySet>
  </EntitySet>

  <GoalSet>
  </GoalSet>
</Domain>
```

One of the ways nodes in BTs communicate with each other is by using the execution *context*: a node may write a variable into the context and another node may use it later. In this scenario it is therefore very important that nodes know the name of the variables that other nodes put into the context. In the set of actions and conditions above there are several nodes that manipulate the context. In particular:

- The *ComputeCharacterPosition* action writes into the context a variable of type *COORDINATE* containing the current position of the unit. The name of such variable is *CharacterPosition*.
- The *ComputeClosestBasePosition* action writes into the context a variable of type *COORDINATE* containing the position of the closest base. The name of such variable is *ClosestBasePosition*.
- The *ComputeRandomClosePosition* action writes into the context a variable of type *COORDINATE* containing a random position that is close to the input position. The name of such variable is *RandomClosePosition*.
- The *LowDanger* sensor writes into the context a variable of type *ENTITY_ID* containing the identifier of the closest dangerous enemy. The name of such variable is *LowDangerTarget*.

A.4. Step 2: Implementing Low Level Actions and Conditions

Once that low level actions and conditions have been defined (see section A.3), the next step is to provide an implementation for them. JBT does not know what does an action such as *ComputeCharacterPosition* or *AttackMove*. Therefore, it is the user of the framework who has to tell JBT how they work.

The life cycle of actions and conditions of a BT is very simple: initially, when the flow of execution reaches the node, it is *spawned*. From then on, at every **game tick**, the node is *ticked*. Every time the node is ticked, it has to report about its termination status, so that the tree may evolve in case the node has finished. As a result, the programmer will have to define how all the domain dependent actions and conditions behave when they are spawned and ticked.

Actions and conditions are each represented by two classes of the JBT Core, *jbt.model.task.leaf.action.ModelAction.java* and *jbt.execution.task.leaf.action.-ExecutionAction.java* in the case of actions, and *jbt.model.task.leaf.condition.-ModelCondition.java* and *jbt.execution.task.leaf.condition.ExecutionCondition.-java* in the case of conditions. Domain dependent actions such as the ones defined above must extend these classes in order for JBT to know how to work with them.

In JBT there are two classes for every type of node. Remember from section A.2.2 that in JBT there are two types of BTs, the *Model BT* and the *Execution BT*. The Model BT is composed of *model tasks*[4], while the Execution BT is composed of *execution tasks*. It is the execution tasks that define how the tasks work, that is, how they behave when they are spawned and ticked. In the case of actions and conditions, the four classes presented above are the base classes for their respective representation as model tasks and execution tasks.

JBT Core offers a tool that semi-automatize the task of creating all the classes from each action and condition of the MMPM domain file. It is the Java class *jbt.tools.btlibrarygenerator.ActionsAndConditionsGenerator.java*.

For every MMPM action, ActionsAndConditionsGenerator creates two classes: one extending ModelAction, which conceptually represents the action, and another one extending ExecutionAction, which represents how the action actually works -whose abstract methods must be completed in order for the action to perform any task at all. We will explain this later-.

Also, for every MMPM boolean sensor, two classes are created: one extending ModelCondition, which conceptually represents the condition (sensor), and another one extending ExecutionCondition, which represents how the condition actually works -whose abstract methods must be completed in order for the condition to perform any task at all. We will explain this later-.

The syntax of the program is as follows:

```
ActionsAndConditionsGenerator -c configurationFile [-r relativePath] [-o]
```

Where `configurationFile` is an XML file that contains all the information required to run the application. The syntax of such file is:

```
<Configuration>
  <DomainFile>MMPMDomainFile1</DomainFile>
```

[4]Remember that in BT terminology, a task and a node are the same thing.

```
  <DomainFile>MMPMDomainFile2</DomainFile>
  ...
  <DomainFile>MMPMDomainFileN</DomainFile>

  <ModelActionsPackage>Name of the package for generated model
  action classes</ModelActionsPackage>

  <ModelConditionsPackage>Name of the package for generated model
  condition classes</ModelConditionsPackage>

  <ModelActionsOutputDirectory>Name of the directory where model
  actions are created</ModelActionsOutputDirectory>

  <ModelConditionsOutputDirectory>Name of the directory where model
  conditions are created</ModelConditionsOutputDirectory>

  <ExecutionActionsPackage>Name of the package for generated
  execution action classes</ExecutionActionsPackage>

  <ExecutionConditionsPackage>Name of the package for generated
  execution condition classes</ExecutionConditionsPackage>

  <ExecutionActionsOutputDirectory>Name of the directory where
  execution actions are created</ExecutionActionsOutputDirectory>

  <ExecutionConditionsOutputDirectory>Name of the directory where
  execution conditions are created</ExecutionConditionsOutputDirectory>
</Configuration>
```

The order in which the elements are specified is not relevant. If the input files do contain only actions, parameters related to conditions may not be specified, and vice versa.

The -r option is used to add a path to the beginning of the files listed in the configuration file; as a result, each file is considered to be placed at the path specified in the -r option. The -r option may not be specified, in which case the files are considered to be at the current execution directory.

The -o option (standing for *overwrite*) is either is specified or not. If it is not specified, generated output files will not overwrite any existing file in the file system, and as a result, the corresponding class file will not be produced in case there is a file with the same name in the file system. If the option -o is specified, then generated output files will overwrite any file in the file system whose name matches.

So, in brief, this program parses a MMPM domain file and, for each action and boolean sensors produces the JBT classes that are required to run such actions and conditions in a BT. In particular, the created execution classes define what they will do when spawned and ticked.

The generated ModelAction and ModelCondition classes are complete, so they do not need to be modified after being created by the ActionsAndConditionsGenerator. However, the ExecutionAction and ExecutionCondition generated classes contain a set of abstract method that must be implemented according to the semantics of the respective actions and conditions, so that

they do what they are expected to do. This is the only step that must be done in order for JBT to be able to work with the low level actions and conditions provided by the user.

In particular, the abstract methods that must be implemented are:

```
protected void internalSpawn();
protected Status internalTick();
protected void internaTerminate();
protected void restoreState(ITaskState state);
protected ITaskState storeState();
protected ITaskState storeTerminationState();
```

internalSpawn() and *internalTick()* are the most important methods, so they should be well implemented.

internalSpawn() represents the spawning process of the task (action or condition). When the flow of execution of the tree reaches the task, *internalSpawn()* gets called. Therefore, this method must be defined so that it starts the process associated to the task. For instance, the *internalSpawn()* method of the *Move* action above should order the current unit to go to the target position; the *internalSpawn()* method of the *Attack* action above should order the current unit to attack the target enemy.

The automatically generated skeleton contains an initial implementation of the *internalSpawn()* method, which is as follows:

```
protected void internalSpawn() {
  /*
   * Do not remove this first line unless you know what it
   * does and you need not do it.
   */
  this.getExecutor().requestInsertionIntoList(
    jbt.execution.core.BTExecutor.BTExecutorList.TICKABLE,
    this);

  /* TODO: this method's implementation must be completed. */
  System.out.println(this.getClass().getCanonicalName() +
  " spawned");
}
```

This initial definition contains a very important aspect of the execution process of the task. As it is said in the comments, the first line should not be removed unless the user knows what it does and he thinks it is not necessary to do it. What that line does is to request that the task be inserted into the list of tickable nodes (section A.2.1). Since in general this is what we want the task to do (because we want the task to receive ticks), **that line should not be removed**.

When implementing all these abstract methods, the user may access the execution context of the task by calling *this.getContext()*. That method just returns the context of the task as an *IContext* object. The *IContext* interface defines two main methods, one for reading a variable from the context, and another one for writing a variable into the context:

```
public interface IContext {
  /**
   * Returns the value of a variable whose name is
   * <code>name</code>, or null if it is not found.
   */
  public Object getVariable(String name);

  /**
   * Sets the value of a variable. If the variable already
   * existed, its value is overwritten. <code>value</code>
   * may be null in order to clear the value of the variable.
   */
  public boolean setVariable(String name, Object value);
  ...
}
```

Remember that MMPM domain files also let the designer specify input parameters for actions and sensors. These are parameters that actions and sensors are supposed to use when running. The generated JBT ExecutionAction and ExecutionCondition classes include *getter* methods for such input parameters. As a result, the programmer will be able to access the value of the input parameters in the abstract methods he has to implement, by using the getter methods. The value for the input parameters are either retrieved from the execution context(IContext) or directly provided at construction time, but these details are hidden from the programmer that implements the action or condition (he should just use the getter methods to retrieve whatever input parameters may be needed). For instance, for the *Attack* action, the next getter method is created:

```
/**
  * Returns the value of the parameter "target", or null in
  * case it has not been specified or it cannot be found in
  * the context.
  */
public java.lang.String getTarget(){...}
```

Thus, in the implementation of all the abstract methods, the user should use this *getTarget()* method if he wanted to retrieve the identifier of the target unit to attack. This whole thing about the getter methods may be a little bit confusing at first. However, bear in mind that when the user creates a behaviour tree (which we will explain later), he can either specify that the input parameter of a task must be retrieved from a variable of the context, or provide an actual value for the parameter. When the task is spawned and run, it does not really care about where the parameter comes from as long as there is a value that it can use. The generated getter methods hide these details, so no matter where the parameter comes from (either from the context or from an actual value provided by the user when he created the BT), it just provides the value that the task expects to use.

Once the task has been spawned, it will be ticked whenever the BT gets ticked. The ticking process is performed by the *internalTick()* method. Thus,

from the moment the task gets spawned, at every tick, *internalTick()* will be called.

internalTick() is in charge of keeping track of the termination status of the task. If the task has not finished yet when *internalTick()* is called, then it must return the termination status *Status.RUNNING*. If the task has finished successfully, then the method should return the termination status *Status.SUCCESS*. If the task has finished unsuccessfully, then the method should return *Status.FAILURE*. For instance, the *internalTick()* method of the *Move* action of our Terran Marine example should check if the unit has arrived at the target position. If it has not, then *Status.RUNNING* should be returned. If the unit has arrived at the target position, then *Status.SUCCESS* should be returned. Finally, y for some reason the action could not be completed (for instance because the target position is unreachable), then *Status.FAILURE* should be returned. Note that, even though the *Status* enum has more values other than *SUCCESS*, *FAILURE* and *RUNNING*, the *internalTick()* method must return only one of these three. Doing otherwise will throw an exception.

One of the ideas behind the *driven by ticks* architecture is that, when the tree is ticked, it should not take very long for it to return, so *internalSpawn()* and *internalTick()* are supposed to return very quickly. However, sometimes tasks perform computationally expensive processes. In that case, instead of performing the expensive computation inside the *internalSpawn()* method of the task, it should be performed in another execution thread. When *internalSpawn()* is called, it should create another thread that carries out the computation. On the other hand, the *internalTick()* method would query the thread to check if its computation has finished or not, and return *Status.RUNNING*, *Status.SUCCESS* or *Status.FAILURE* accordingly.

The *internalTerminate()* method is not so important. Sometimes, when a BT is running, some of its tasks get abruptly interrupted. This happens for instance when one of the children of a parallel task (which is following the *sequence policy*) fails. When that happens, all of its children, which were being concurrently evaluated, get terminated, so they must stop running. Other scenario in which this happens is when a perform interruption task interrupts an interrupter. In that case, the interrupter and its child stop running.

It is for cases like these that the *internalTerminate()* method is defined. The *internalTerminate()* method must make the task stop running. For instance, in the case of the *Move* action, it should order the unit to stop moving. Moreover, if the task started some other thread to perform some computations, it should stop it. In general, when the *internalTerminate()* is called, the task should stop running and should also free whatever resources it acquired.

With respect to the *storeState()*, *storeTerminationState()* and *restoreState(ITaskState state)*, they are related to *persistent tasks*. Some tasks in BTs are persistent in the sense that, after finishing, if they are spawned again, they should remember past information. Take for example the Limit task. A Limit task allows to run its child node only a certain number of times (for example, 5). After being spawned, it has to remember how many times it has been run so far, so that, once the threshold is exceeded, it fails. In general, it could be said that some tasks need to retain some persistent information to be used in the future when the task is spawned again.

All the persistent information of a task should be saved into an *ITaskState* object. The ITaskState interface represents a collection of variables that can

be accessed:

```
public interface ITaskState {
  /* Returns the value of a state variable by its name. */
  public Object getStateVariable(String name);
}
```

When a task finishes, (it returns *Status.SUCCESS* or *Status.FAILURE* in *internalTick()*), the *storeState()* method is automatically called by the framework. This method should then create and return an *ITaskState* object containing all the persistent information that the task may need if it is spawned again in the future. If no persistent information is needed, then *null* must be returned. An *ITaskState* object is just a collection of variables that can be retrieved by name. In order for the user to create *ITaskState* objects, he can make use of the factory class *jbt.execution.core.TaskStateFactory*.

It also may be the case that a task that is abruptly terminated (*internalTerminate()* is called) needs to store persistent information. If that is the case, then *storeTerminationState()* is automatically called by the framework, instead of *storeState()*. *storeTerminationState()* follows the same semantics as *storeState()*.

The persistent information that a task stores via its *storeState()* and *storeTerminationState()* methods is restored via the *restoreState(ITaskState state)* method. This method is called just before the task is spawned (that is, before calling *internalSpawn()*). In *restoreState(ITaskState state)* the task should analyse the input *ITaskState* object and restore whatever information it contains (if *null*, then it means that there is no past information to remember).

In the case of the *Move* action of the Terran Marine example, the *storeState()*, *storeTerminationState()* and *restoreState(ITaskState state)* methods are empty.

As a final example on implementing low level actions and conditions, lets follow the whole process for the few actions and conditions defined in the domain explained in section A.3.

Let us suppose that the MMPM domain file is stored in the file *TerranMarineDomain.xml*. Then, the ActionsAndConditionsGenerator application could be run with the next configuration file (stored in *configurationFile.xml*):

```
<Configuration>
  <DomainFile>TerranMarineDomain.xml</DomainFile>
  <ModelActionsPackage>bts.actions</ModelActionsPackage>
  <ModelConditionsPackage>bts.conditions</ModelConditionsPackage>
  <ModelActionsOutputDirectory>src/bts/actions
  </ModelActionsOutputDirectory>
  <ModelConditionsOutputDirectory>src/bts/conditions
  </ModelConditionsOutputDirectory>
  <ExecutionActionsPackage>bts.actions.execution
  </ExecutionActionsPackage>
  <ExecutionConditionsPackage>bts.conditions.execution
  </ExecutionConditionsPackage>
  <ExecutionActionsOutputDirectory>src/bts/actions/execution
  </ExecutionActionsOutputDirectory>
  <ExecutionConditionsOutputDirectory>src/bts/conditions/execution
```

```
  </ExecutionConditionsOutputDirectory>
</Configuration>
```

Let us suppose that the ActionsAndConditionsGenerator is called with the following arguments:

```
ActionsAndConditionsGenerator -c configurationFile.xml -r
                                          /home/outputDirectory
```

Then, the ActionsAndConditionsGenerator will parse the domain file */home/-outputDirectory/TerranMarine.xml*, and it will create output classes for each action and boolean sensor in the domain file, which will be stored in the corresponding files. For instance, for the *Attack* action, two classes will be created, */home/outputDirectory/src/bts/actions/Attack.java* (the model task class) and */home/outputDirectory/src/bts/actions/execution/Attack.java* (the execution task class). For the boolean *LowDanger* sensor, two classes will be created, */home/outputDirectory/src/bts/conditions/LowDanger.java* (the model task class) and */home/outputDirectory/src/bts/conditions/execution/-LowDanger.java* (the execution task class). Then we should implement the abstract methods of all the execution classes. For instance, the implementation of the *Move.java* execution class may be like this:

```
/** ExecutionAction class created from MMPM action Move. */
public class Move extends ExecutionAction {
  ...
  /**
   * Returns the value of the parameter "target", or null if
   * not found anywhere.
   */
  public float[] getTarget() {
    /* Whatever has been automatically generated. */
    ...
  }

  protected void internalSpawn() {
    this.getExecutor().requestInsertionIntoList(
    jbt.execution.core.BTExecutor.BTExecutorList.TICKABLE,
    this);

    /*
     * Retrieve the identifier of the entity (Terran Marine)
     * running this action from the context. Here we are
     * assuming that the context will contain it.
     */
    String currentEntityID = (String) this.getContext().
                             getVariable("CurrentEntityID");

    /*
     * Now we assume that there is a generic "Game Engine"
     * that can be used to send generic orders to units of
     * the game. Note that, in order to retrieve the target
```

```
     * position, we use the automatically generated getter
     * method.
     */
    GameEngine.sendMoveOrder(currentEntityID,
    this.getTarget());
  }

  protected jbt.execution.core.ExecutionTask.Status
                                          internalTick() {
    /*
     * In this method we will just check whether the unit
     * has reached the target position. If the target
     * position is unreachable, then Status.FAILURE is
     * returned. Otherwise, if the unit has reached the
     * target position, Status.SUCCESS is returned.
     * Otherwise, Status.RUNNING is returned.
     */
    String currentEntityID = (String) this.getContext().
                             getVariable("CurrentEntityID");

    if(!Util.reachablePosition(currentEntityID,
                                    this.getTarget())){
      return Status.FAILURE;
    }

    float[] currentPosition = GameEngine.getPosition(
                                          currentEntityID);
    float[] targetPosition = this.getTarget();

    if(Math.distance(currentPosition, targetPosition)
                                                  < 0.5){
      return Status.SUCCESS;
    }
    else{
      return Status.RUNNING;
    }
  }

  protected void internalTerminate() {
    /* We just order the unit to stop. */
    String currentEntityID = (String) this.getContext().
            getVariable(Constants.CONTEXT_CURRENT_ENTITY);

    GameEngine.stopUnit(currentEntityID);
  }

  protected void restoreState(
                    jbt.execution.core.ITaskState state) {
    /* Does nothing. */
  }
```

```
  protected jbt.execution.core.ITaskState storeState() {
    /* No persistent state information to return. */
    return null;
  }

  protected jbt.execution.core.ITaskState
                                  storeTerminationState() {
    /* No persistent state information to return. */
    return null;
  }
}
```

A.5. Step 3: Creating BTs with JBT Editor

Once that the domain dependent actions and conditions of the game have been defined and a JBT implementation for them has been provided, the next steps are quite easy to follow.

It is now when the user should define the behaviour trees to use in the game. Following our Terran Marine example, we will define several trees that implement the behaviour that we described in section A.3.

Behaviour trees are first described in XML files. Here we are not going to describe the XML format that JBT understands, since we discourage the user from writing them in plain text. Instead, we propose to use the JBT Editor (composed of the two projects under the "./JBTEditor" directory of the repository), a GUI application through which it is really easy to define behaviour trees.

The JBT Editor is an Eclipse RCP application. As such, it must be run under the Eclipse SDK environment or use the executable files provided for each platform. When opening if from Eclipse, you have to open the project *jbt.tools.bteditor*, and then open the file *bteditor.product* with the Product Configuration Editor. Once it has been opened, go to the "Overview" page, and click on "Launch an Eclipse application". A window like that of figure A.4 should show up.

The JBT Editor is a very simple tool, so learning how to use it should not take very long.

In order to create a new BT, just click on the "new BT" icon or select "File-¿New BT". A new editor should open up showing an empty BT (that is, a tree containing only a single node, the root of the tree. To the right of the window there is a tree-like menu (the *Nodes Navigator*) where the user can select nodes to build the tree. In order to add a node to a tree, just drag it from the Nodes Navigator and drop it onto whatever node of the tree to insert it as a child or sibling of the target node. The root node can have only one child. However, other nodes, such as sequences or selectors may have many of them. Decorators can have only one child, and leaf nodes do not have any children.

By using the set of provided standard nodes of the Nodes Navigator, the user can build complex behaviour trees. However, in order to build really useful trees, the user must use the domain-dependent actions and conditions from the game. The JBT Editor lets the user load a MMPM domain file as described

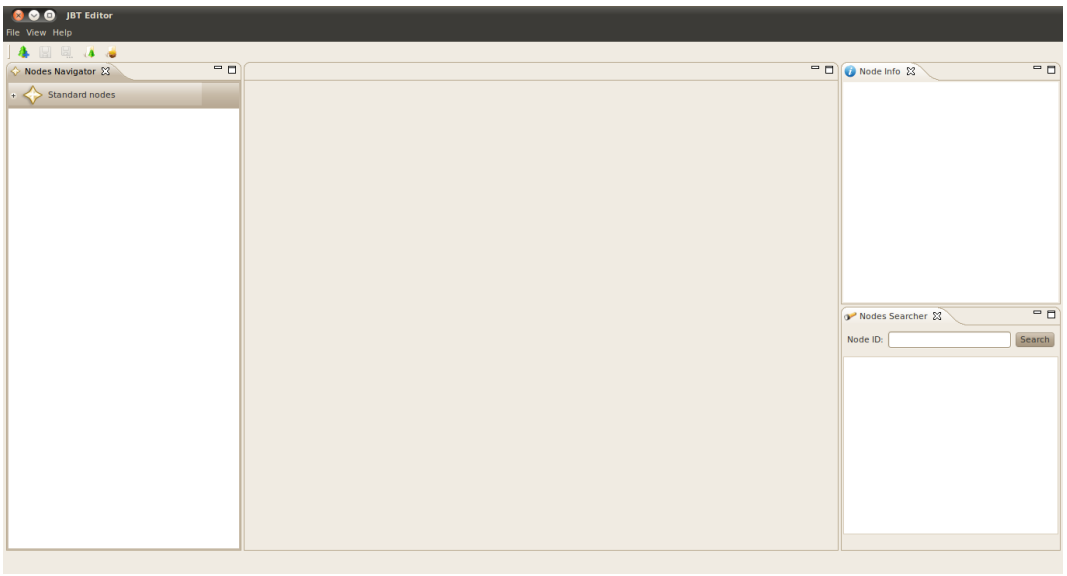

Figura A.4: JBT Editor after being opened

in section A.3. Just click on the "Load MMPM Domain" or select "File-¿Load MMPM Domain" and select the file that contains the low level actions and conditions. After doing so, the Nodes Navigator will be added a new entry for the actions and conditions within the domain file, which the user will be able to use when building BTs.

For instance, if we load the domain file described in section A.3, we get the actions and conditions shown in figure A.5.

The JBT Editor lets the user specify values for the input parameters of nodes. By double clicking on a node that has input parameters, a dialog where the user can specify values for the input parameters. For instance, if the AttackMove action is double clicked, then the dialog of figure A.6 is shown. In the dialog the user can specify a value for the parameter "target", whose type is *COORDINATE*. Thus, the text field only supports values such as "45 62" or "10 -3 45". As we mentioned in section A.4, when building a BT, the user can specify whether the input parameters of actions and conditions are retrieved from the context or an actual value is provided at construction time. The "From context" check box lets the user specify if the parameter has to be retrieved from the context or not. If the check box is not ticked, then the user must provide a value for the parameter in the text field. However, if the check box is ticked, then what the user does is to specify the name of the context's variable where the value of the parameter will be retrieved from.

For instance, in figure A.7 the user has specified a value ("12 34 4.5") for the input parameter "target" of the AttackMove action. However, in figure A.8 the user has indicated that the value of the input parameter "target" will be retrieved from the variable "TargetVariable" of the context.

Once the BT has been completed, the user can save it as an XML file. In order to do so, just click on the "Save" or "Save As" icons (or select "File-¿Save" or "File-¿Save As") and enter a file name.

When saving a BT, the JBT Editor checks if the structure of the tree is correct. If not, incorrect nodes are highlighted in red color and an explanation for the error is shown in the *Node Info* view (to the right of the window) when the node is selected.

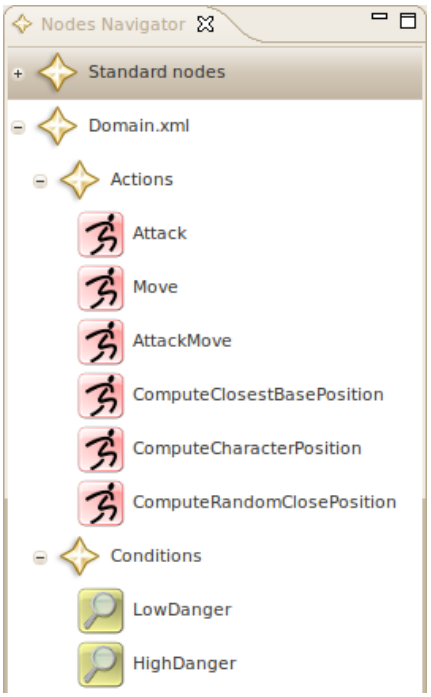

Figura A.5: The Nodes Navigator after loading the domain file

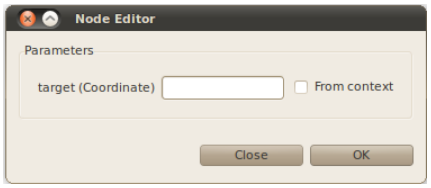

Figura A.6: The dialog for editing the input parameters of the AttackMove action

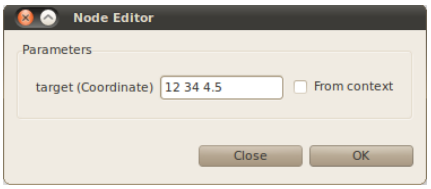

Figura A.7: A parameter for which an actual value is provided

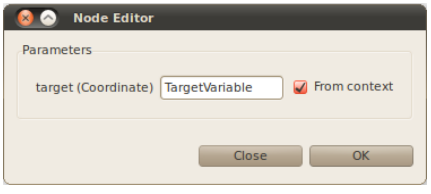

Figura A.8: A parameter whose value will be retrieved from the context

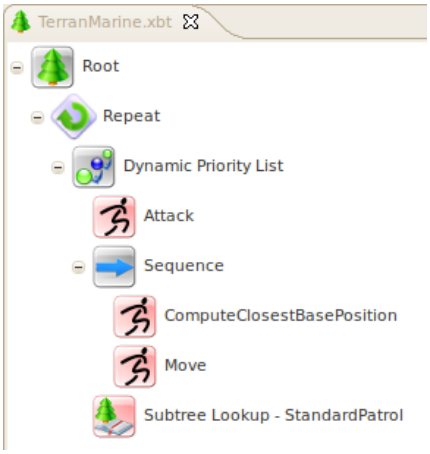

Figura A.9: Initial tree for the Terran Marine behaviour

Also note that a name for the BT must be provided. The name of a BT is specified in the root node of the tree. When the root is double clicked, a dialog appears that lets the user assign the tree a name. BTsñames are very important, because it is the way that trees are referenced. In particular, names are essential in terms of reusability. For instance, the *Subtree Lookup* task simulates a particular BT, and the way that the Subtree Lookup knows what BT to simulate is by providing the name of the tree.

Let us now design the BT that implements the Terran Marine behaviour that we described in section A.3. An initial implementation for such tree could be that of figure A.9.

The behaviour is pretty simple: the tree is always executing (see the *Repeat* node) a *Dynamic Priority List* (DPL) that is constantly checking three conditions:

- If the current unit is in a low danger situation, then the unit is ordered to attack the closest dangerous enemy. This is represented by the first child of the DPL.
- If the current unit is in a high danger situation, then the unit runs away to the closest base. This is represented by the second child of the DPL (the Sequence).
- Finally, if none of the above conditions are met, the unit just "patrols". This is represented by the third child of the DPL, the Subtree Lookup node (we will further explain this later).

There are some details that must still be defined though. So far, the tree does not provide a way of checking the three conditions above. In order to do so, we can make use of guards, since the DPL interacts with children that have guards defined. In order to add a guard to a node, just right click on the node and select "Edit Guard". A dialog should appear that lets the user edit the guard of the node.

In JBT, guards are represented by BTs. In particular, a guard can be a single node or a complete and correct BT. When the user clicks on "Add simple guard", a dialog lets the user select a single leaf node (action, condition or a standard leaf node) to be used as a guard. If the user clicks on the "Add

Figura A.10: Selecting a guard for the *Attack* node

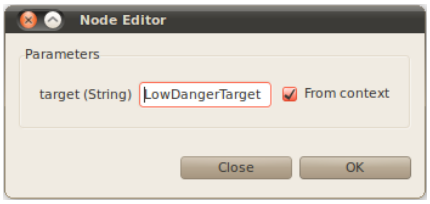

Figura A.11: The input parameter of *Attack*

complex guard", however, a new editor is opened so the user can create a complete BT to be used as the node's guard.

In our example, the guard of the *Attack* action is just the condition *LowDanger*. Therefore, we have to click on the "Add simple guard" button and select the "LowDanger" task from the list, as shown in figure A.10. Note that after inserting the guard, a small shield icon will appear on top of the node that has been added a guard.

In the case of the *Sequence* node, its guard is just the condition *HighDanger*, so we add it just the same way. Now we have to define the input parameters of the tasks.

First of all there is the *Attack* node. The intended behaviour is for the soldier to attack the closest unit once that the *LowDanger* condition has been triggered. Thankfully, *LowDanger* writes into the context the identifier of the closest entity (as we described in section A.3), in a variable with name *LowDangerTarget*. Therefore, the input parameter of *Attack* should be as represented in figure A.11, since the value of the input parameter of the *Attack* action is present in a variable of the context whose name is *LowDangerTarget*.

With respect to the *Move* action, the idea is that the unit goes to the closest base. Since the position of the closest base has been computed by the *ComputeClosestBasePosition* action and written into the context in a variable with name *ClosestBasePosition*, then the input argument of *Move* must be read from the context and its value must be the variable name *ClosestBasePosition*.

Now let us look at the final part of the tree. When none of the danger conditions are met, the marine has to patrol around its current position. Since *patrolling* is a complex task that may be reused in another trees, we decide to put it into another BT and reuse of from the Terran Marine BT. This is accomplished by the Subtree Lookup task, whose input parameter is set to *StandardPatrol*. *StandardPatrol* is the name of the tree that will implement the patrol behaviour.

The tree of figure A.12 implements the patrol behaviour. Initially, the current position of the unit is computed by the *ComputeCharacterPosition* ac-

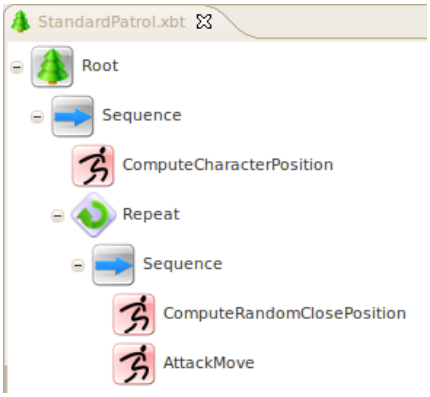

Figura A.12: The BT for the Standard Patrol behaviour

tion. This position is written into the variable *CharacterPosition* of the context, as we mentioned in section A.3. From then on, the is a forever loop (Repeat node) that constantly computes a random position that is close to the one computed by the *ComputeCharacterPosition* task, and then orders the unit to *AttackMove* to that target position. It is important to note that the tree must have a name (it is set in the root of the tree), in this case *StandardPatrol*, which is the name that was used in the *SubtreeLookup* task of the Terran Marine BT.

Note that a name for the Terran Marine BT must be provided, so we will assume that its name is *TerranMarine*.

A.6. Step 4: Creating a Java Declaration of the BTs

Once our BTs have been defined in the XML format of the JBT Editor, the next steps are quite easy. Actually, there is no more complex work to be done by the user from now on.

So far we have provided a definition and implementation of domain dependent low level actions and conditions. We have also defined our BTs and stored them in XML files using the JBT Editor. The next step is to provide a Java implementation of the trees so that JBT can actually run them.

This step is automatically performed by the JBT Core. The JBT Core has an application, the *jbt.tools.btlibrarygenerator.BTLibraryGenerator.java*, that basically takes the XML definition of some BTs and creates a .java file that contains the implementation of such trees.

In JBT, BTs are grouped together in BT libraries. A BT library is just a collection of BTs that can be retrieved by name (actually, the name that is specified for the tree in the JBT Editor). A BT library is implemented by the *IBTLibrary* interface. This interface just represents a set of BTs that can be retrieved by name:

```
public interface IBTLibrary extends
                        Iterable<Pair<String, ModelTask>> {
    /* Returns the BT of name name, or "null" if not found. */
    public ModelTask getBT(String name);
}
```

Note that, in JBT, a Model BT (see section A.2.2) is represented by the abstract class *ModelTask*. Thus, when we ask an *IBTLibrary* to give us a BT by its name, it just retrieves the Mode BT that represents the tree, which is represented by a *ModelTask*.

What the BTLibraryGenerator does is to automatically create a BT library, that is, a class that implements the *IBTLibrary* interface, that can be used to retrieve the BTs defined in the XML files. In particular, given a set of behaviour trees specified in XML files and the MMPM definition of the low level actions and conditions that are used in the trees, it creates the corresponding Java class.

The syntax of this program is as follows:

```
BTLibraryGenerator -c configurationFile [-r relativePath] [-o]
```

Where `configurationFile` is an XML file that contains all the information required to run the application. The syntax of such a file is:

```
<Configuration>
  <BTLibrary>
    <BTFile>BTFile1</BTFile>
    <BTFile>BTFile2</BTFile>
    ...
    <BTFile>BTFileN</BTFile>

    <DomainFile>MMPMDomainFile1</DomainFile>
    <DomainFile>MMPMDomainFile2</DomainFile>
    ...
    <DomainFile>MMPMDomainFileN</DomainFile>

    <ModelActionsPackage>Name of the package where model action
    classes are placed</ModelActionsPackage>

    <ModelConditionsPackage>Name of the package where model
    condition classes are placed</ModelConditionsPackage>

    <LibraryClassName>Name of the class that is going to be
    created</LibraryClassName>

    <LibraryPackage>Name of the package for the generated BT
    library</LibraryPackage>

    <LibraryOutputDirectory>Name of the directory where the
    generated library is going to be stored</LibraryOutputDirectory>
  </BTLibrary>

  <BTLibrary>
  ...
  </BTLibrary>

  ...
</Configuration>
```

The order in which the elements are specified is not relevant.

In the file the user can define several BT libraries, each one within the *BTLibrary* element. For each BT library defined in a *BTLibrary* element, the program will produce an output file (class implementing the *IBTLibrary* interface) for the library.

The -r option is used to add a path to the beginning of the files listed in the configuration file; as a result, each file is considered to be placed at the path specified in the -r option. The -r option may not be specified, in which case the files are considered to be at the current execution directory.

The -o option (standing for overwrite) is either is specified or not. If it is not specified, the generated output files will not overwrite any existing file in the file system, and as a result, a behaviour tree library may not be produced in case there is a file with the same name in the file system. If the option -o is specified, then generated output files will overwrite any file in the file system whose name matches.

In the Terran Marine example, we want to create a BT library that contains the two BTs that we created in section A.5. Let us suppose that the tree defining the Terran Marine behaviour was stored in the file *TerranMarine.xbt*, and that the tree defining the patrol behaviour was stored in the file *StandardPatrol.xbt*. Then, we could use the following configuration file to produce the BT library:

```
<Configuration>
  <BTLibrary>
    <BTFile>TerranMarine.xbt</BTFile>
    <BTFile>StandardPatrol.xbt</BTFile>
    <LibraryClassName>TerranMarineBTLibrary</LibraryClassName>
    <LibraryPackage>bts.btlibrary</LibraryPackage>
    <LibraryOutputDirectory>src/bts/btlibrary</LibraryOutputDirectory>
    <DomainFile>TerranMarineDomain.xml</DomainFile>
    <ModelActionsPackage>bts.actions</ModelActionsPackage>
    <ModelConditionsPackage>bts.conditions</ModelConditionsPackage>
  </BTLibrary>
</Configuration>
```

Where note that *TerranMarineDomain.xml* is the domain file that we created in section A.3. The *ModelActionsPackage* and *ModelConditionsPackage* must also be those specified in the configuration file of the ActionsAndConditionsGenerator.

So now let us suppose that the configuration file above is stored in a file called *BTConfigurationFile.xml*. The BTLibraryGenerator may be called with the following arguments:

```
BTLibraryGenerator -c BTConfigurationFile.xml -r
                                        /home/outputDirectory
```

Then, the BTLibraryGenerator will parse the configuration file and it will realize that it has to create just one BT library (the only *BTLibrary* element in the configuration file). It will then parse the XML files of the trees that the library will contain (*TerranMarine.xbt* and *StandardPatrol.xbt*) and finally will create an BT library class named *TerranMarineBTLibrary*, which will be

placed in the output directory *home/outputDirectory/stc/bts/btlibrary*. Our purpose here is not to analyse all the details of the produced class, but just point out that it can be used through the public interface that it implements (the *IBTLibrary* mentioned earlier). The class *TerranMarineBTLibrary* will contain both trees, so

```
getBT("TerranMarine");
```

will return the tree implementing the Terran Marine behaviour, and

```
getBT("StandardPatrol");
```

will return the tree implementing the patrol behaviour.

The way we can actually run these trees is explained in section A.7.

A.7. Step 5: Running the Behaviour Trees

So that is almost all. The last step is to run the trees that have been put into one or several BT libraries.

The way BTs are run in JBT is really simple, and follows the ideas mentioned in sections A.2.1, A.2.2 and A.2.3.

Basically, the IBTLibrary is used to retrieve Model BTs. Once a Model BT has been retrieved, a BT Executor must be created to run it. The BT Executor must be fed with an initial context that the BT will use when running[5]. Let's suppose that we have created our *TerranMarineBTLibrary*, and that we want to use it in order to control a particular Terran Marine in the game. First of all, two details must be taken into account.

On the one hand, the actions that we implemented (well, we actually implemented only one action, *Move*, but you get the idea) made the assumption that the identifier of the unit running the action was present in the context, under a variable named "CurrentEntityID" (you can revisit the implementation in section A.4). In general, it may be necessary for the context that is used by the tree to have some initial variables in it. In such case, an appropriate context should be created.

On the other hand, the *SubtreeLookup* task poses a big problem when it is run. Remember that the *SubtreeLookup* node simulates the behaviour of a tree given its name. However..., how does it know from where to retrieve a tree given its name? For instance, in the case of the Terran Marine tree, the *SubtreeLookup* is suppose to emulate a tree named *StandardPatrol*, but it does not know where to get such a tree from.

The context is used in order to fix this problem. Actually, the context must contain all the trees that are used within the internal execution of the tree. In this case it means that the context that is initially passed to the tree must contain the BT *StandardPatrol*. If it does not, when the Terran Marine tree is run it will throw an exception complaining about not being able to find the corresponding tree.

We can see that an appropriate context must be created to run the tree. JBT Core provides a factory class, the *jbt.execution.core.ContextFactory.java*

[5]Remember that the BT that actually runs is the Execution BT, but that detail is of no interest at this point.

that defines several methods for creating generic contexts (*IContext* objects of the Basic Context type described in section A.2.4.1). In our case, we can make use of the method that takes as input an *IBTLibrary* and makes an *IContext* that contains all the BTs of the BT library. Then, we could make use of the methods in the *IContext* interface to add the identifier of the marine that is supposed to be run by the tree:

```
/* First of all, we create the BT library. */
IBTLibrary btLibrary = new TerranMarineBTLibrary();

/*
 * Then we create the initial context that the tree will
 * use.
 */
IContext context = ContextFactory.createContext(btLibrary);

/*
 * Now we are assuming that the marine that is going to be
 * controlled has an id of "terranMarine1"
 */
context.setVariable("CurrentEntityID","terranMarine1");
```

The next step is to create the BT Executor to run the tree. In JBT, a BT Executor is represented by the *IBTExecutor* interface. In order to create an *IBTExecutor* object to run a particular BT, the factory class *jbt.execution.core.BTExecutorFactory.java* must be used. The *BTExecutorFactory* has several methods for creating BT Executors; one of them receives as input the Model BT to run and the initial context:

```
/* Now we get the Model BT to run. */
ModelTask terranMarineTree = btLibrary.getBT("TerranMarine");

/* Then we create the BT Executor to run the tree. */
IBTExecutor btExecutor = BTExecutorFactory.createBTExecutor(
                                        terranMarineTree, context);

/* And finally we run the tree through the BT Executor. */
do{
  btExecutor.tick();
}while(btExecutor.getStatus() == Status.RUNNING);
```

Note that running a BT is a very simple process. The *IBTExecutor* interface defines one main method, *tick()*, which implements the ticking process of a BT. Every time the *tick()* method is called, the BT is given some CPU time to do its work, as was explained in section A.2.1.

In order to check the current execution status of the tree, the *getStatus()* method is used. As long as the status is *Status.RUNNING*, the tree has not finished so it should continue to receive ticks.

Bibliografía

[1] Agnar Aamodt and Enric Plaza. Case-based reasoning; foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59, 1994.

[2] Perry Alexander and Costas Tsatsoulis. Using sub-cases for skeletal planning and partial case reuse. *Int. J. Expert Syst.*, 4(2):221–247, 1991.

[3] William M. Bain. Judge: a case-based reasoning system. pages 1–4, 1986.

[4] Ray Bareiss. *Exemplar-Based Knowledge Acquisition*. Academic Press, 1989.

[5] Blizzard. http://us.blizzard.com/en-us/games/sc/.

[6] Robin Douglas Burke. *Representation, storage and retrieval of tutorial stories in a social simulation*. PhD thesis, Evanston, IL, USA, 1993.

[7] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189 – 208, 1971.

[8] Gonzalo Flórez-Puga, Marco A. Gómez-Martín, Pedro P. Gómez-Martín, Belén Díaz-Agudo, and Pedro A. González-Calero. Query enabled behaviour trees. *IEEE Transactions On Computational Intelligence And AI In Games*, 1(4):298–308, November 2009.

[9] Damian Isla. Halo 3 - building a better battle. In *Game Developers Conference*, 2008.

[10] Bobby D. Bryant Kenneth O. Stanley and Risto Miikkulainen. Real-time neuroevolution in the nero video game. *IEEE Transactions on Evolutionary Computation*, pages 653–668, 2005.

[11] Janet Kolodner. *Case-based reasoning*. Morgan Kauffmann, first edition, 1993.

[12] John Krajewski. Creating all humans: A data-driven AI framework for open game worlds. *Gamasutra*, February 2009.

[13] Gun Ho Lee. Rule-based and case-based reasoning approach for internal audit of bank. *Know.-Based Syst.*, 21(2):140–147, 2008.

[14] Team Liquid. http://wiki.teamliquid.net/starcraft/chaoslauncher.

[15] Beatriz López and Enric Plaza. Case-based planning for medical diagnosis. In *ISMIS '93: Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems*, pages 96–105, London, UK, 1993. Springer-Verlag.

[16] R.V. Magaldi. R.v. cbr for troubleshooting aircraft on the flightline. In *Proceedings of the IEE Colloquium on Case-Based Reasoning: Prospects for Applications*, 1994.

[17] MakeMEPlayME. http://sourceforge.net/projects/darmok2/.

[18] Cynthia R. Marling and Peter Whitehouse. Case-based reasoning in the care of alzheimer's disease patients. In David W. Aha and Ian Watson, editors, *4th International Conference on Case-Based Reasoning, ICCBR 2001, Proceedings*, pages 702–715, 2001.

[19] Ian Millington and John Funge. *Artificial Intelligence for Games*. Morgan Kaufmann, second edition, 2009.

[20] Héctor Muñoz-Avila, David W. Aha, Dana S. Nau, Rosina Weber, Len Breslow, and Fusun Yamal. Sin: integrating case-based reasoning with task decomposition. In *IJCAI'01: Proceedings of the 17th international joint conference on Artificial intelligence*, pages 999–1004, 2001.

[21] Dana S. Nau, Yue Cao, Amnon Lotem, and Hector Mu noz-Avila. Shop: Simple hierarchical ordered planner. In *IJCAI '99: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 968–975, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[22] Trung Nguyen, Mary Czerwinski, and Dan Lee. Compaq quicksource: Providing the consumer with the power of artificial intelligence. In *IAAI '93: Proceedings of the The Fifth Conference on Innovative Applications of Artificial Intelligence*, pages 142–151. AAAI Press, 1993.

[23] Santiago Onta nón, Kane Bonnette, Prafulla Mahindrakar, Marco A. Gómez-Martín, Katie Long, Jainarayan Radhakrishnan, Rushabh Shah, and Ashwin Ram. Learning from human demonstrations for real-time case-based planning. *IJCAI-09 Workshop on Learning Structural Knowledge From Observations*, 2009.

[24] Santiago Ontañón, Kane Bonnette, Prafulla Mahindrakar, Marco A. Gómez-Martín, Katie Long, Jainarayan Radhakrishnan, Rushabh Shah, and Ashwin Ram. Learning from human demonstrations for real-time case-based planning. In Ugur Kuter and Héctor Muñoz-Avila, editors, *Proceedings of the IJCAI-09 Workshop on Learning Structural Knowledge From Observations*, 2009.

[25] Santiago Ontañón, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. On-line case-based planning. *Computational Intelligence*, 26(1):84–119, 2010.

[26] Marc Ponsen, Pieter Spronck, Héctor Muñoz-Avila, and David W. Aha. Knowledge acquisition for adaptive game ai. *Science of Computer Programming*, 67(1):59 – 75, 2007. Special Issue on Aspects of Game Programming.

[27] Jim Prentzas and Ioannis Hatzilygeroudis. Categorizing approaches combining rule-based and case-based reasoning. *Expert Systems*, 24(2):97–122, 2007.

[28] Gonzalo Flórez Puga, Marco Antonio Gómez-Martín, Belén Díaz-Agudo, and Pedro A. González-Calero. Dynamic expansion of behaviour trees. In *AIIDE*, 2008.

[29] A. Ram and J. C. Santamariá. Continuous case-based reasoning. *Artif. Intell.*, 90(1-2):25–77, 1997.

[30] E. L. Rissland and K. D. Ashley. A case-based system for trade secrets law. In *ICAIL '87: Proceedings of the 1st international conference on Artificial intelligence and law*, pages 60–66, New York, NY, USA, 1987. ACM.

[31] Barry Smyth and Mark T. Keane. Design à la déjà vu - reducing the adaptation overhead, 1996.

[32] Pieter Spronck, Marc Ponsen, Ida Sprinkhuizen-Kuyper, and Eric Postma. Adaptive game ai with dynamic scripting. *Mach. Learn.*, 63(3):217–248, 2006.

[33] Kenneth O. Stanley. Evolving neural network agents in the nero video game. In *In Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games*, pages 182–189, 2005.

[34] Neha Sugandh, Santiago Ontañón, and Ashwin Ram. Real-time plan adaptation for case-based planning in real-time strategy games. In Klaus-Dieter Althoff, Ralph Bergmann, Mirjam Minor, and Alexandre Hanft, editors, *Advances in Case-Based Reasoning*, volume 5239 of *Lecture Notes in Computer Science*, pages 533–547. Springer Berlin / Heidelberg, 2008.

[35] BWAPI Team. http://code.google.com/p/bwapi/.

[36] Ben Weber. http://eis.ucsc.edu/starproxybot.

Autorización

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Extendiendo Darmok, un Sistema de Planificación Basada en Casos, mediante Árboles de Comportamiento", realizado durante el curso académico 2009-2010 bajo la dirección de Pedro Antonio González Calero en el Departamento de Ingeniería del Software e Inteligencia Artificial, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Fdo: Ricardo Juan Palma Durán
En Madrid a 13 de setiembre de 2010